拡張の合成に適したメタレベルの構成

田中 哲　　渡部 卓雄
Akira Tanaka　　Takuo Watanabe
akr@jaist.ac.jp　　takuo@jaist.ac.jp
北陸先端科学技術大学院大学

概 要

メタレベルアーキテクチャにおいては，メタレベルを拡張することによってベースレベルの記述言語の意味を変更するが，一般にメタレベルの拡張は複数存在し，それらの相互作用によって予期しない結果が生じることがある．このような問題を防ぐためには，メタレベルの拡張の方法に制約を加えて，相互作用が予測しやすい形で行なわれるよう保証する必要がある．本稿では，複数の拡張を合成して同時に適用することが容易にになるよう，そのような拡張方法について述べる．この方法では，抽象構文木として表現されたベースレベルに属性を付与することによって意味づけを行なう．このような形のメタレベルは言語処理系に限らない非常に広範囲に適用でき，本稿ではλ計算に似た簡単な言語および LR パーザのメタレベルを例としてあげる．

1 はじめに

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アーキテクチャである．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．例えば，拡張可能な言語処理系では言語処理系自体とその拡張がメタレベルとなり，その拡張を利用して記述されたプログラムがベースレベルとなる．

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．

- 用途に適した記述法を使えるためベースレベルの記述が単純になる．
- 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．

例えば，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，言語仕様上の制限[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る．一般に，ある機能を実現する記述を集中できるほどモジュール化が容易であるため，メタレベルアーキテクチャはモジュール化を支援するということができる．ここで，メタレベルモジュール（メタレベルの拡張モジュール）がメタレベル内の他の部分とどのように相互作用するか，また，メタレベルモジュールをどのようにメタレベルに挿入するかなど，メタレベルを拡張するためのインターフェースをメタインターフェースと呼ぶ．

メタレベルの拡張はメタインターフェースを通して行なわれるため，拡張の行ないやすさや拡張自体をモジュール化できるかどうかは，メタインターフェースの設計に依存する．しかし，このようにメタレベルの善し悪しの鍵となるにもかかわらず，メタインターフェースの設計についてはとくに決まった指針がなく，個々のメタレベル実装毎にさまざまな設計が行なわれている．

メタインターフェースは，通常の拡張可能なプログラムと基本的には同様のものである．つまり，外部からカスタマイズ可能なパラメータがあったり，プログラムの断片を適当な場所に挿入できたりするというものである．例えば，CLOS[2] にはメソッドサーチのクラス優先順位リストを指定するパラメータがあり，また，メタクラスの定義時には既存のメタクラスを継承した上でメソッドを再定義することができる．また，3–Lisp[4] では，自己反映手続きの呼び出しにより，環

[1] 例えば，記述内の変数の型が違うなど．

境や継続などのパラメータを参照することができる．

ここで，どのようなパラメータを用意するか，またプログラムの断片をどこに追加するかなどといった具体的な点は用途に依存する．例えば，CLOS のクラス優先順位リストはクラスベースのオブジェクト指向言語でしか意味を持たず，それ以外のメタレベルには無意味である．一般に（メタレベル以外のプログラムでも），用途が具体的であるほど，カスタマイズが可能であるべき部分（メタレベル開発者の一存では決定できない部分）とそれ以外の部分が明確に分離でき，拡張のインターフェースを適切に設計できる．しかし，メタレベルというだけではメタレベルというものがベースレベルの意味を定義するということしか決まっていないためメタインターフェースを設計することは簡単ではない．

既存のメタインターフェースの設計は大きく分けて次のように分類できる．

- アプリケーションにあわせて設計されたもの

  これは特定のアプリケーションの分野を仮定し，その分野に必要な拡張を実現できるようにメタインターフェースを定義したものである．この方法は用途（アプリケーション）を具体的に仮定できるため，適切なメタインターフェースが定義しやすいという利点がある．しかし，仮定した以外の用途には使いにくいメタインターフェースになりがちである．

- 言語処理系にあわせて設計されたもの

  これは言語処理系の内部構造の中で拡張の容易な部分をアクセスできるようにメタインターフェースを定義したものである．この方法は用途を仮定しないため，一般的な（特定のアプリケーションの概念を含まない）メタインターフェースを定義できる．しかし，定義したメタインターフェースが広範囲なアプリケーションに適用可能かどうかは必ずしもあきらかではない．

つまりメタインターフェースを定義するのに参考になるほど具体的な構造はアプリケーションや処理系の内部構造が主で，メタレベルアーキテクチャ自体にはないのである．

本稿では拡張に適したメタレベルをつくるための一般的な指針を示す．そのために，2 節でさまざまなメタレベルに共通する構造をあげ，その構造にアクセスするメタインターフェースの設計について述べる．3, 4 節ではそのメタインターフェースを使った具体的なメタレベルを述べる．6 節で関連研究について述べ，7 節でまとめを行なう．

2 メタレベルの構成

本稿ではアプリケーションや処理系の内部構造に依存しない具体的な構造として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．

この仮定はプログラムをテキストとして表現するほとんどの言語について成り立つ．この仮定をおくと，メタレベルに次のような拡張を考えることができる．

- 新しい構文を導入する．
- 既存の構文の意味を変更する．

つまり，それぞれの構文の意味をメタレベルが定義すると考えるため，構文単位の拡張を考えることができるのである．

メタレベルは拡張を前提としたプログラムであるから，安全に拡張できるようになっていなければならない．このために拡張をモジュール化し不測の相互作用を避けられるような構成とする．そのために，まずベースレベルの意味をメタレベルが定義する方法について 2.1 節で述べ，メタレベルのモジュールとその相互作用について 2.3, 2.2 節で述べる．

2.1 ベースレベルの意味

ベースレベルの抽象構文木が $G = (N, T, P, S)$ という文法に従うとする．ここで N, T, P は非終端記号，終端記号，導出規則の集合であり，S は開始記号である．また導出規則（P の要素）は次の形をしている．

$$X_0 \to X_1 \ldots X_n \quad (n \geq 1)$$

ただし $X_0 \in N, X_1 \in T, X_j \in N (1 < j \leq n)$ である．以下，$p, p', \ldots$ で P の要素を表現する．ここで，右辺の先頭に必ず終端記号が現れるのは抽象構文だからで

あり，この終端記号は導出規則を一意に同定するものとする．つまり，

$$\begin{aligned} & p = X_0 \to X_1 \ldots X_n \wedge \\ & p' = X'_0 \to X'_1 \ldots X'_{n'} \wedge X_1 = X'_1 \\ & \quad \Rightarrow \quad n = n' \wedge X_0 = X'_0 \wedge \ldots \wedge X_n = X'_n \end{aligned}$$

が成り立つとする．また，導出規則内に現れた同一の記号 X の出現位置を後から区別するために添字を右肩につけて X^i と記述することがあるが，この場合右肩の添字の違いは記号自体の違いとはみなさない．

プログラムの意味は属性文法的に構文木上の属性として表現する．属性には名前があり，構文木の各頂点に複数の（異なる名前の）属性がつけられる．属性の値は評価関数（メタレベル）によって計算され単一代入によって属性に結びつけられる．ここで，属性は構文木の根だけでなくすべての頂点につき得るが，プログラム全体の意味は根の属性によって与えられる．それ以外の頂点の属性は部分プログラムの意味などを表現する．また，属性文法に対して次のような拡張を行なう．

- 属性評価時に動的に構文木を生成し，既存の構文木に接ぎ木する．（高階属性文法 [8]）
- 動的に生成する構文木の部分木として既存の構文木を利用する [5].

これらにより，ベースレベルの構文木に沿わない構造の計算を構文木上で表現することが可能となる．後者の場合，既存の構文木をその内部の非終端属性に接ぎ木した場合などには構文木がサイクルを含む一般のグラフになることもあり得る．

ベースレベルの意味をメタレベルが定義するということは，各頂点につく属性の種類や計算方法をメタレベルが定義するということである．つまり，メタレベルは属性とその評価規則の定義からなり，メタレベルによって意味を求める場合には，属性の依存関係に従って評価規則を実行して根の属性を求めることになる．

メタレベルは (A, B, R, N, T, P, S) の 7 つ組である．ここで (N, T, P, S) は抽象構文木の文法，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る．また 評価規則（R の要素）は導出規則と評価式の組であり，次のように表記する．

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

ここで p は導出規則であり $p = X_0 \to X_1 \ldots X_n$ 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．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関数とする．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），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.

- a : X_0 の属性 a を示す.
- b : X_0 の非終端属性 b を示す.
- $X_j.a$: ここで $0 < i \leq n$ であり，X_j 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_j : ここで $0 \leq i \leq n$ であり，X_j 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.

また，評価規則の先頭の p を実際に記述することが繁雑な場合には省略して代わりに X_1 と記述する．

なお，形式的には根の属性の値がプログラムの意味となるが，実際のプログラムの実行はその値を実行時環境と組み合わせて行なうことになる．例えば，プログラムの意味を変数環境から最終結果の値への関数として定義すれば，メタレベルによって求められた関数に変数環境を適用することによってプログラムを実行することになる．

2.2 メタレベルモジュール

メタインターフェースはメタレベルを変更するためのインターフェースであり，メタレベルにその差分を追加するものである．そこで本節では差分とその追加方法を定義する，

メタレベルの拡張はメタレベル中の A, B, R, N, T, P に対する変更とし，メタレベルモジュール ΔM を次のように定義する．

$$\Delta M \quad = \quad (\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P)$$

ここで ΔA, ΔB, ΔR, ΔN, ΔT, ΔP は A, B, R, N, T, P に対する追加である．

つまり，メタレベル $M = (A, B, R, N, T, P, S)$ に ΔM を拡張してできた新しいメタレベル M' は次のようになる．

$$\begin{aligned} M' &= (A \cup \Delta A, B \cup \Delta B, R \cup \Delta R, \\ &\quad N \cup \Delta N, T \cup \Delta T, P \cup \Delta P, S) \end{aligned}$$

また，メタレベルモジュールはそれぞれの要素を和集合とすることによって合成できる．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．

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

where

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

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．

$$(\Delta M, S) = (\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P, S)$$

where $\Delta M = (\Delta A, \Delta B, \Delta R, \Delta N, \Delta T, \Delta P)$

2.3 メタレベルの相互作用

メタレベルは 7 つ組 (A, B, R, N, T, P, S) として表現されるが，実際の処理は R 中の評価規則で定義され，その相互作用は属性の代入と参照によって行なわれる．ここで，属性は単一代入であるため，相互作用は単純な依存関係になり，しかも関係を同定する明瞭な方法（属性名）があるという把握しやすいものになる．

また，R 以外の A, B, N, T, P, S は実行可能なプログラムが含まれていない静的なものなので，相互作用は名前の衝突程度しか起きない．名前の衝突は名前空間の導入によって容易に避けられるのでこれは大きな問題にはならない．

3 簡単な言語のメタレベル

本節では値呼び λ 計算に相当する簡単な言語 L のメタレベル M_{L} を実際に構成することにより，基本的なメタレベルの構成法を述べる．

3.1 共通部分の定義

λ 計算の文法は式 E という非終端記号とシンボル S という終端記号を持つ．シンボルは属性 label : L に名前を保持しているとする．ここで L は名前の型である．そして，式の意味を属性 eval : $(L \to V) \to V$ によって定義する．ここで V はベースレベルで扱う値の型である．これらの属性だけをもつメタレベルモジュールを図 1(a) のように ΔM_{Base} というモジュールとして定義する．

3.2 基本構文の定義

λ 計算の基本的な構文は変数参照 Var，関数適用 App，関数抽象 Abs の 3 種類である．従って，それぞれの構文の定義を 図 1(b), 図 1(c), 図 1(d) のように $\Delta M_{\mathrm{Var}}, \Delta M_{\mathrm{App}}, \Delta M_{\mathrm{Abs}}$ というモジュールとして定義する．

ΔM_{Base} に $\Delta M_{\mathrm{Var}}, \Delta M_{\mathrm{App}}, \Delta M_{\mathrm{Abs}}$ を合成し，開始記号を E とすると，基本的な λ 計算の解釈を行なうメタレベル M_{L} ができる．M_{L} は図 1(e) のようになる．

3.3 メタレベルの拡張

ここで，L に対する拡張として任意変数の参照を許すメタレベルモジュール (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}) について述べる．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 3.4 節で述べる。

任意変数の参照

ここで，簡単な拡張として，任意の変数を参照する構文を拡張するモジュールを考える．この構文は式を引数としてとる．実行時にはその式を評価し，その結果を変数名とみなして環境から値を取り出し，その値

$$\Delta M_{\text{Base}} = (\{\text{eval}, \text{label}\}, \phi, \phi, \{E\}, \{S\}, \phi)$$

(a) 共通部分

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

(b) 変数参照

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

(c) 関数適用

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

(d) 関数抽象

$$\begin{aligned}\Delta M_{\text{L}} &= (\{\text{eval}, \text{label}\}, \phi, \{r_{\text{Var}}, r_{\text{App}}, r_{\text{Abs}}\} \\ &\quad \{E\}, \{S, \text{Var}, \text{App}, \text{Abs}\}, \\ &\quad \{p_{\text{Var}}, p_{\text{App}}, p_{\text{Abs}}\}) \\ M_{\text{L}} &= (\Delta M_{\text{L}}, E)\end{aligned}$$

(e) 合成されたメタレベル

図 1: 簡単な言語のメタレベル

を返す.

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

この例は，メタレベルがベースレベルよりも自由に計算状態にアクセスできることを示している．つまり，基本的な λ 計算では実現不可能な機能を拡張している．

自由変数を求める

また，より複雑な拡張として自由変数を求める構文を考える．この構文は式を引数としてとり，実行時にはその式の自由変数の集合を返す.

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

この例では，メタレベルモジュールががそれ自身で部分式をトラバースしている．そのために他のモジュールで定義した構文に対する評価規則を追加している．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 [6].

Let

また，非終端属性を使う拡張として 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．

$$\Delta M_{\text{Let}} = (\phi, \{L\}, \{r_{\text{Let}}, q_{\text{Let}}\}, \phi, \{\text{Let}\}, \{p_{\text{Let}}\})$$

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

ここで，構文木を動的に生成するためにそれぞれの導出規則に対応する構成子が存在して評価関数の中から利用可能であるとする．

この定義では，Let 式は非終端属性 L をもち，L には 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，L 上での計算結果を 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．

3.4 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 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 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 LR パーザのメタレベル

本節では狭義の言語以外のメタレベルの例として LR パーザの例を取り上げる．LR パーザは BNF で定義された文法とそれに付随したアクションをから，その文法に従って入力を構文解析するものである．

LR パーザの基本的な動作は次の 4 段階にわかれる．

1. 入力から item の集合を求める．
2. LR オートマトンを求める．
3. 衝突を解決する．
4. パーザを生成する．

例えば，次のような文法定義に対して生成される item の集合と（衝突を算術式に適切なように解決した）LR オートマトンは例えば 図 2 のようになる．

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

4.1 メタレベルの概要

メタレベルは動作の段階毎にモジュール化され，4 つのメタレベルモジュール ΔM_{Item}，ΔM_{At}，ΔM_{Res}，ΔM_{Parse} によって定義される．そして，これら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} が次のように定義される．

$$\Delta M_{\text{LR}} = \Delta M_{\text{BNF}} + \Delta M_{\text{Item}} + \Delta M_{\text{At}}$$

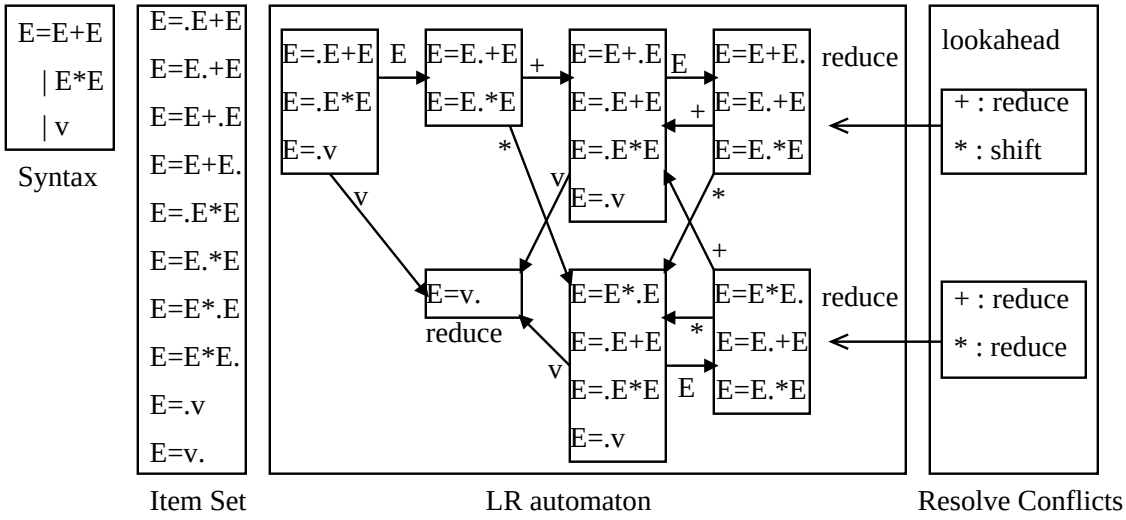

図 2: LR パーザの生成例

$$
\begin{aligned}
& + \quad \Delta M_{\text{Res}} + \Delta M_{\text{Parse}} \\
M_{\text{LR}} \quad &= \quad (\Delta M_{\text{LR}}, g)
\end{aligned}
$$

ここで，g は ΔM_{BNF} で定義される開始記号である．

文法記述の文法

入力は BNF であり，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} となる．ここで，メタレベルが扱う文法 (BNF 自体) とベースレベルで扱う文法 (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．

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

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．

item の集合を求める

item は導出規則の右辺のどこかに印 (.) 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ように定義できる．

$$
\begin{aligned}
\Delta M_{\text{Item}} \quad = \quad & (\{\text{items, lhs, left, right}\}, \\
& \phi, R, \phi, \phi, \phi)
\end{aligned}
$$

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 は次の評価関数からなる．

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

$$
\begin{array}{rcl}
\text{Syms} & : & ss.\text{left} := \text{left} +\!\!+ [s.\text{label}] \\
\text{Syms} & : & \text{right} := [s.\text{label}] +\!\!+ ss.\text{right} \\
\text{Syms} & : & \text{items} := ss.\text{items} \cup \{\text{lhs} = \text{left.right}\} \\
\text{Sym0} & : & \text{right} := [] \\
\text{Sym0} & : & \text{items} := \{\text{lhs} = \text{left.}\}
\end{array}
$$

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

LR オートマトンを求める

LR オートマトンは item の集合の部分集合が状態と対応するようなオートマトンである．従って初期状態から到達可能なすべての状態に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 と呼ばれる．subset construction が生成する状態数は 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]．従ってこの段階の処理は基本的に根の評価関数内で行ない，結果を構文木として動的に生成する．

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

ここで R は次の評価関数からなる．

Gram : A := items からオートマトンを求める

また $\Delta N_{\text{At}}, \Delta P_{\text{At}}$ はオートマトンを表現するための構文規則であり，次のように定義される．

$$
\begin{array}{rcl}
\Delta N_{\text{At}} & = & \{st, bs, b\} \\
\Delta P_{\text{At}} & = & \{st \rightarrow \text{State } bs \\
 & & \ bs \rightarrow \text{Branch0} \\
 & & \ bs \rightarrow \text{Branches } b\ bs \\
 & & \ b \ \rightarrow \text{Shift } s\ st \\
 & & \ b \ \rightarrow \text{Reduce } a\}
\end{array}
$$

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．

[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．

衝突を解決する

一般に，LR パーザで衝突が起こった場合，衝突した分岐のど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方法 [1] 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，多種多様な方法がある．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するため，次のようなメタレベルで実現できる．

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

R は次の評価規則からなる集合である．

$$
\begin{array}{rcl}
\text{Shift} & : & \text{applicable} := \text{Shift 可能かどうか} \\
\text{Reduce} & : & \text{applicable} := \text{Reduce 可能かどうか}
\end{array}
$$

ここで，applicable は引数として計算状態を受けとって真偽値を返すものとし，その真偽値によってその分岐が選択可能であるかどうかが判断される．このように衝突の解決のインターフェースだけを決めることにより，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．

構文解析を行なう

LR オートマトンを入力のトークン列に適用することによって構文解析を行なうことができる．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．この計

算状態はまた衝突解決に利用される．このような構文解析は次の ΔM_{Parse} によって実現できる．

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

ここで，R は次の評価規則からなる．

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

5 メタレベルの合成

メタレベルモジュールは一般に（属性名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．

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

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 $(\Delta M_{\text{LLR}}, E)$ 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 $(\Delta M_{\text{LLR}}, g)$ 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．

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

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．

6 関連研究

言語などの拡張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，拡張に対する基本的な見方を提供している．しかし，本文でも触れたとおり，メタレベルアーキテクチャ自身は具体的な拡張法を提供しているわけではない．そのため，メタレベルアーキテクチャにしたがったさまざまな方法が提案されている．

6.1 リフレクション

リフレクション [9] 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．このため，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，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，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例外的である．従ってベースレベルとメタレベルは異なる対象を扱うことが多く，それぞれ固有の性質を無視して同じ記述系を使用するのは必ずしも合理的ではない．

6.2 AOP

AOP[3] はプログラムの複数の側面を別々に記述し，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．これは，ある側面の記述の意味はその側面だけからは決まらず，他の側面の記述によっても変化するという意味で，個々の側面が他の側面の言語を拡張していると見なすことができる．

AOP では各側面間の関係は Aspect Weaver が定義するというものであるため，各側面はベースレベル的な記述であることもあれば，メタレベル的な記述（他の側面の記述の意味を補完するもの）であることもある．つまり，AOP は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ったメタレベルアーキテクチャとも見なせる．

7 おわりに

本稿では拡張の容易なメタレベルを構成するためのメタインターフェースについて考察した．メタインターフェースはメタレベルを操作するためのインターフェースであり，その設計はメタレベルの善し悪しの鍵になるにもかかわらず，良い設計をするには適用範囲を狭めなければならないというジレンマを持つ．本稿では，アプリケーションにも言語処理系の内部構造にも依存しない広範な適用範囲を持ち，また，メタレベル内部での相互作用が明解なメタインターフェースを設計する方法を示した．この相互作用の明解さはメタレベル内のモジュールの組み合わせをベースレベルの構文木というユーザの目に見える形に沿って行なうことと，モジュール間の相互作用が単一代入という単純な形で行なわれることによる．そして，その上でいくつかの具体的なメタレベルを構成しそのことを実証した．また，高階属性文法によって既存の構文を使った構文木を動的に生成することにより再利用性が実現できることも確認した．また，LR パーザの例では衝突の解決部分がモジュールとして実現でき入れ換えることが容易であることも述べた．

本稿では，例として値呼びの λ 計算に相当する言語と，LR パーザについて触れたが，本稿で述べたメタレベルの構成法は「ベースレベルの記述が抽象構文木として表現される」という仮定しかおいていないので，言語処理系とは関係のない他の応用にも利用可能である．例えば，(オブジェクト指向言語での）クラス階層や，XML[7] を対象にするメタレベルも考えられる．

参考文献

[1] D. Grune and C. J. H. Jacobs. *PARSING TECHNIQUES A Practical Guide*. Ellis-Horwood, 1990.

[2] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect–oriented programming. In *Proceedings ECOOP '97*, LNCS 1241, pp. 220–242. Springer Verlag, June 1997.

[4] B. C. Smith. Reflection and semantics in Lisp. In *ACM POPL*, pp. 23–35, Jan. 1984.

[5] A. Tanaka and T. Watanabe. An extensible LR parser generator — a case study of composable metalevel extensions —. In *Proceedings of International Workshop on Principles of Software Evolution*, pp. 84–88, July 1999.

[6] 田中, 渡部. 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, Sept. 1999.

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

[8] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, 24(7):131–145, July 1989.

[9] 渡部. リフレクション. コンピュータソフトウェア, 11(3):5–14, May 1994.