構文拡張可能なオブジェクト指向言語の設計と試作

93B567 森 忠夫 93B582 野村 和孝 93B589 奥村 三四郎

指導教員　野呂 昌満

1　はじめに

現在のソフトウェア開発は複数人数での共同開発が主流となっている．しかし、現存するプログラミング言語の実行時環境が統一されていないので、開発に用いるプログラミング言語を開発者個人個人が、異なるものを使用することは難しい．この問題を解決する方法として、言語の構文を拡張可能にすることが考えられる．言語の構文が拡張可能であれば、その言語の構文を複数の異なる言語の構文に拡張することで共通の実行時環境を複数の言語で共有できる．これまでは、処理系の変更やライブラリやマクロ定義を利用した機能の拡張での構文拡張が考えられていた．しかし、これらの方法による拡張には以下のような問題点がある．

- 利用者の手作業に依存する部分が大きい
- 拡張できる機能が限定される

これらの問題点を解決し、柔軟な言語拡張を可能とする手段として自己反映計算がある．我々の提案する言語同様、オブジェクト指向言語に自己反映計算を取り入れる研究は過去に行なわれているが、構文の変更を目的としたものは少なく、また構文の変更が可能であっても、変更可能な構文が一部に限定されているといった欠点がある．我々は、

- 新構文パターンを文脈自由文法として
- 新構文で使用する字句を正規表現として

定義することでこれらの問題点が解決できると考えた．しかし、構文は使用者により動的に定義されるため、定義したパターンを生成規則として新たな構文解析・字句解析部を生成する手法が必要となる．その第一段階として本年度は、増殖的に字句解析部を生成することを目標とし、自己反映機能を持つ字句の追加・変更が可能なオブジェクト指向言語 *y-1* の設計・試作を行なった．また、*y-1* の字句解析部の実現に用いた方法と同様の方法で、構文解析部の実現も可能であると考えている．我々は複数の異なる言語を使用したソフトウェアの共同開発を実現する手段として、構文拡張可能なプログラミング言語の実現を最終目標としている．

2　関連研究

2.1　自己反映計算

計算システムが、自分自身の構造や計算過程を計算システム自身が操作することを自己反映計算 (reflection)[7] という．自己反映計算の能力を持つシステムを自己反映的 (reflective) なシステムと呼ぶ．システムが自己反映的であるためには以下の条件を満たしていなければならない．

1. システムが、自分自身の構造や計算過程をモデル化した表現 (自己表現 /self representation) を内部に保持する．
2. システムと自己表現が互いに他を反映している関係 (因果的結合の関係 /causal connection) にある．

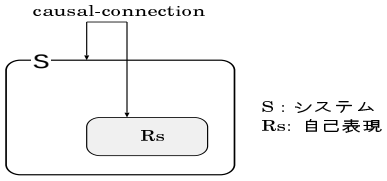

図 1: 自己反映的なシステム

手続き的リフレクション (procedual reflection)

自己反映システムを実現する一般的な方法として知られている手続き的リフレクション [7] について述べる．手続き的リフレクションとは、計算をベースレベルとメタレベルに分離して考えることで自己反映計算を実現する方法である．手続き的リフレクションに基づいたオブジェクト指向プログラミング言語処理系では応用プログラムを記述する次元をベースレベル、応用プログラムの自己表現を解釈実行する次元をメタレベルとする．

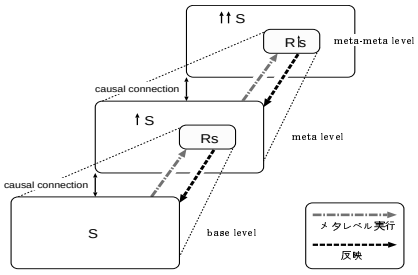

図 2: リフレクティブタワー

手続き的リフレクションに基づく処理系では、メタレベルとベースレベルとを同一の言語で記述するので、無限のリフレクティブタワーが構築される (図 2参照)．実際には無限の階層を構築することは不可能であるので、処理系を動作させるための、ユーザが操作することができない核を用い、有限の階層として実現するのが普通である．

2.2 拡張可能なオブジェクト指向言語

過去に研究された拡張可能なオブジェクト指向言語について調査した．ここでは、MPC++[4]、OpenC++[1]、RbCl[9]、y-0[8] の特徴を以下に述べる．

MPC++

MPC++ はメタアーキテクチャを利用者に公開することで、言語の構文を動的に拡張することのできる言語である．メタアーキテクチャとしては、字句解析オブジェクト、名前表オブジェクト、構文解析オブジェクト、構文木オブジェクト、コード生成オブジェクト、といったメタレベルオブジェクト群から構成される抽象コンパイラと、それらのメタオブジェクトを変更するための操作を定義している．抽象コンパイラのオブジェクト群を利用者が変更することで構文を拡張できる．

MPC++ は、構文拡張のための構文木クラスの記述と応用プログラムの記述が異なるという点で、自己反映アーキテクチャでない．

OpenC++

OpenC++ は処理系を構成する言語要素を取り扱うためのメタオブジェクトのメソッドを再定義することで言語の拡張を実現している．メタオブジェクトは関数・クラスの単位で構成され、それらに対応する目的コードを変更することにより意味の拡張を行なっている．記述性を向上させるための構文拡張が可能であるが、型名、クラス名、new 演算子の修辞子 (modifier) の記述に限定されている．OpenC++ は個別構成法に基づく自己反映オブジェクト指向言語といえる．OpenC++ はクラス・関数単位でメタオブジェクトを生成し、目的コードを変更するので複数のクラスや関数に共有する構文の変更は不可能である．

OpenC++ で拡張可能な構文は先に述べたとおり修辞子に限定されており、本研究の目的である任意の拡張を行なうことは OpenC++ のアーキテクチャでは不可能である．

RbCl

RbCl は、群構成法で実現されており、実行時カーネルを持たない自己反映並列オブジェクト指向言語である．RbCl は、実現言語との言語的共生を導入し、実行時カーネルを持たずにすべての実行時ルーチンを変更可能である．

この拡張方法は柔軟な拡張を可能にしているが、新オブジェクトのコードをすべて記述する必要があるので拡張は容易ではないと考えられる．

y-0

y-0 ではメタレベルインタプリタに対してメッセージを送ることにより構文の拡張を行なっている．メタレベルインタプリタは構文変更のメッセージを受けとると、構文テーブルに新たに構文を登録する．登録後、使用者はその構文を使用してプログラムを記述することが出来る．メタレベルインタプリタは、構文テーブルに登録された構文のパターンに基づき原始プログラムを解釈し、ベースレベルインタプリタで解釈可能な基本構文に変換する．

y-0 でのマクロ展開方式による拡張の実現は容易だが、

- 拡張構文を区切り記号で囲まなければならない
- 構文パターンに繰り返し、省略をもちいることができない
- メタオブジェクトプロトコルとして公開されているのは、構文テーブルに対する操作だけである

などの問題点があり、任意の拡張が困難である．

2.3 ISG(Incremental Scanner Generator)

ISG は、字句解析部において字句の追加・削除を動的に行なうことができる処理系である．Lex などの字句解析部生成系は字句の追加を前提としていないので、状態遷移表の構築や最適化に必要とされる前処理の時間は考慮されていない．なぜなら Lex などの生成系では、生成された実行効率のよい状態遷移表を数多く使うことによって前処理にかかる時間が清算されると考えているからである．しかし、ISG は動的な字句の追加・削除を行うことを目的としているので、字句の変更を行うさいの前処理の時間を短くしなければならない．この問題を解決するために ISG では状態遷移表の再構築のさいに、字句を追加する前の状態遷移表を利用することで、前処理の時間を短縮している．

アルゴリズム

ISG の提案するオートマトンの構築及び字句の追加によるオートマトンの再構築のアルゴリズムについて述べる．

- *CONSTRUCT/SCAN*
  CONSTRUCT は正規表現の集合から決定性有限オートマトン (DFA) を構築する．
  SCAN は構築された DFA で入力文字列の照合を行なう．
- *L-CONSTRUCT/L-SCAN*
  L-CONSTRUCT は *DFA* の初期状態を構築する．
  L-SCAN は文字列の照合を行いながら正規表現の集合から部分的なオートマトン (*PDFA*[1]) を構築する．

[1] 完全に構築されていないオートマトンを *PDFA*(Partial-DFA) と呼ぶ

- *MODIFY/S-MODIFY*
  *MODIFY*は、 *L-CONSTRUCT/L-SCAN*で構築された *PDFA* を利用して、新たに追加された正規表現を認識できる *PDFA* を再構築する.
  また *S-MODIFY*は、 *PDFA* の最適化を行う
- *EXPAND/CC-EXPAND*
  *EXPAND*は *CONSTRUCT/SCAN*や *L-CONSTRUCT/L-SCAN*において、正規表現の集合から状態の遷移先を求めるアルゴリズムである.
  CC-EXPAND は正規表現において文字の集合を一つの要素とする文字クラス (character class) を認識できるように *EXPAND*を拡張したものである

MODIFY

ここでは、増殖的な字句解析部の設計において参考としたアルゴリズム *MODIFY*について述べる. このアルゴリズムは正規表現の追加の際に、追加前のオートマトンを再利用して新たなオートマトンを構築していくものである. 以下に手順を示す.

1. 正規表現の追加があると、新たにオートマトンの初期状態 **S'** を生成する. この初期状態 **S'** は、追加前の正規表現の集合に新たな正規表現を追加した正規表現の集合から計算した初期状態である.
2. この初期状態 **S'** から遷移先を計算する. ここで拡張を行なう前の *PDFA* **X**(図 3) を再利用して、新たな *PDFA* の構築を行なう.
3. 新たに生成した初期状態から遷移できない状態を除去して、拡張された *PDFA* **Y**(図 3) が完成する.

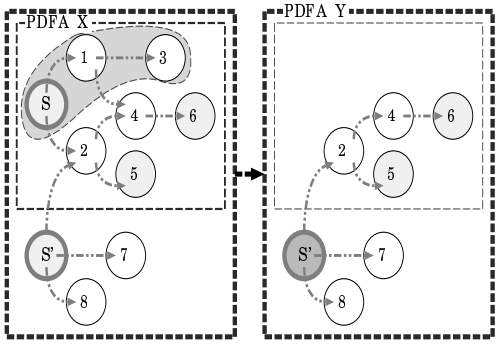

図 3: アルゴリズムの流れ

3 構文拡張可能な言語の構成法

自己反映計算に基づき、構文拡張を行なうための枠組を以下のように提案する.

- 字句解析部、構文解析部などの各解析部を処理系が扱う言語の自己表現と考える.
- 新構文でもちいる字句を正規表現で、新構文パターンを文脈自由文法で定義する.
- 正規表現、文脈自由文法をもとに、増殖的に字句解析部・構文解析部を構築することで、構文の拡張を実現する.
- 言語の構文はその言語で記述されたすべてのオブジェクト群の共有する資源であると考え、この拡張方式による構文拡張を実現するために、処理系を個別構成法ではなく群構成法に基づき設計する.

本研究では、増殖的に字句解析部を生成することを目標としているので、ここでは構文解析部については述べない.

3.1 群構成法 (Group-Wide Architecure)

群構成法とは、個別構成法とは相反する特徴を持った構成法である. 群構成法では、複数のオブジェクトで構成されるグループがメタレベル表現の単位となっている. 複数のベースオブジェクトがオブジェクトグループを構成し、オブジェクトグループに対応するメタオブジェクトグループが存在する. 群構成法ではオブジェクトのグループを単位としているので、ベースオブジェクト群が共有する資源の制御の記述には適している.

3.2 字句拡張方式

我々は自己表現の一部である字句解析部を動的に拡張することで字句の拡張を可能にすると考える. 拡張のための操作は応用プログラムと同様の記述を用い、自己表現へメッセージを送るという方法をとる. 字句は、以下のように定義する.

- 正規表現 (regular-expression)
- アクション (action)

これを自己表現へのメッセージの引数として与える. 正規表現は、字句を構成する文字・記号の並びである. アクションは入力文字列と正規表現が一致した時に起こるものである. 自己表現はこれらを受け取ることにより状態遷移表の再構築を行なう.

4 オブジェクト指向言語 *y-1*

我々の提案する拡張方式に基づく字句解析部が字句の追加・変更が可能であることを示すために、オブジェクト指向言語 *y-1* を設計し、その処理系を試作する. *y-1* は、字句の追加・変更を示すことを目的とし、その言語仕様は簡素化した.

4.1 *y-1* 言語仕様

我々は、以下のように *y-1* 言語仕様を定義した.

- 強く型付けされた言語である.
- 必要最低限の機能しか持たない.
  実行可能なコードは代入文とメッセージ文のみ
  クラスは *TOKEN* クラスを継承したものしか定義できない

クラス

y-1 では利用者が定義できるクラスは *TOKEN* クラスの派生クラスのみである. 以下に *TOKEN* クラスを示す.

```
class TOKEN{
    int TVAL ;
    string TTXT ;
    method:
      New(){}
    int getVal() {
        return TVAL;
    };
    string getText() {
        return TTXT;
    };
} ;
```

また、*y-1* では利用者が使用することのできるクラスとして **int**、**string**、**metaclass**、**rule** を用意している. **int** とは整数型、**string** とは文字列型である. **metaclass** とは、そのインスタンスがクラスを保持する型のことをいう. **rule** とは、正規表現と **metaclass** 型変数を保持し、メソッドに

firstpos 正規表現の最初の記号の集合を返す

lastpos 正規表現の最後の記号の集合を返す

followpos 正規表現で、ある記号の次に来る記号の集合を返す

eval metaclass 型変数が保持しているクラスのインスタンスを生成する

を持つ型のことをいう.

クラス本体

利用者には、クラス本体にコンストラクタの内容の記述のみが許されている. コンストラクタは、オブジェクト生成時に起動されるメソッドである. コンストラクタには戻り値の型はなく、その本体にオブジェクト生成時に行ないたいことを記述する. 本体には、代入文のみ記述できる. 本体は、外部に定義することはできず、クラス定義内に記述しなければならない.

実行部

構文要素 **main** の後にコードを記述する. 実行部は変数宣言部と本体とにわけて記述を行なう. 本体は、構文要素 **begin** からはじめ、**end** で終る. 本体内で許されているコードは、代入文、自己表現 (Self_Representation) へのメッセージと出力文のみである. 以下に記述例を示す.

```
main
    int a;
    metaclass b(A);
    rule c(":=",b);
begin
    a = 1;
    Self_Representation<-addToken(c);
    a := 1;
end
```

4.2 字句拡張

y-1 での字句拡張について述べる. *y-1* では自己反映計算の考えにより自己表現へメッセージを送ることで、字句解析部の拡張を行うことができる. メッセージを受けとった自己表現は、保持している状態遷移表を変更し新たな字句を認識することを可能とする. 自己表現へは、正規表現とアクションとで定義した新字句 (**rule** オブジェクト) を引数としてメッセージを送る.

正規表現

新字句の定義に用いることのできる正規表現は、以下の通りである.

a	アスキーコードからなる記号
\a	* の様なメタ記号から抜け出るもの
ab	連結
a|b	選択
a*	ゼロかそれ以上の繰り返し
(a)	グループ化のための括弧

アクション

自己表現の持つ状態遷移表において入力文字列と正規表現が一致した時に行ないたいことを記述してあるものが、アクションである. *y-1* では、*TOKEN* クラスの派生クラスを利用者が定義し、そのクラスのコンストラクタにアク

ションとして行なうことを記述する． したがってそのクラスのインスタンスを生成することで、アクションが起こる．

```
class A : TOKEN {
    New(){

        token t = new Self_Representation
                    < −lookupToken(“=”);
        TVAL = t<−getVal();
    } ;
} ;
```

メタオブジェクトプロトコル

y-1 では以下に示すメタオブジェクトプロトコルを公開することで字句の追加・変更を可能としている．

Self_Representation<−*addToken*(**rule** オブジェクト)

引数に **rule** オブジェクトを取り、このメッセージを受けとった自己表現は **rule** オブジェクトに対しメッセージを送ることで状態遷移表の変更を行なう．

Self_Representation<−*lookupToken*(” 正規表現”)

戻り値として、引数の正規表現と一致したさいに生成する **metaclass** 型の変数を返す．

5 *y-1* 処理系の設計

我々の提案する字句拡張方式での拡張が可能であることを確認するために *y-1* 処理系を試作した．*y-1* 処理系はインタプリタとして作成した．なぜなら、原始プログラム中で字句拡張のための記述がある場合、その時点でメッセージを解釈実行し、追加された字句をそのあとの記述に反映させるからである．本処理系は、字句拡張が可能であることを確認することが目的なので、実行効率を向上させることについては考慮しない．*y-1* 処理系は字句解析部とインタプリタで構成されており、それぞれ以下に述べる．

5.1 字句解析部

字句解析部は、構文解析部からの要求にしたがい原始プログラムの字句を読み込み、その字句と一致した正規表現のクラスのインスタンスを生成して返す．字句拡張をおこなうさいにはインタプリタからのメッセージにより以下の各オブジェクトを拡張することで新たな字句の認識が可能となる．字句解析部を構成するオブジェクトは

- 字句解析オブジェクト
- 状態遷移表オブジェクト
- 正規表現管理オブジェクト

である．
以下に各オブジェクトについて述べる．

字句解析オブジェクト
字句解析オブジェクトは、状態遷移表オブジェクト、正規表現管理オブジェクト、読み込んだ字句を保持するバッファ、正規表現に一致した文字列を入れるための配列からなる．

状態遷移表オブジェクト
状態遷移表オブジェクトは、状態オブジェクトのリスト、正規表現管理オブジェクトからなる． 各状態オブジェクトは内部に現在の状態からの遷移先に関する情報を保持する． 遷移先は状態オブジェクトである．

正規表現管理オブジェクト
正規表現管理オブジェクトは、 **rule** オブジェクトを保持する．

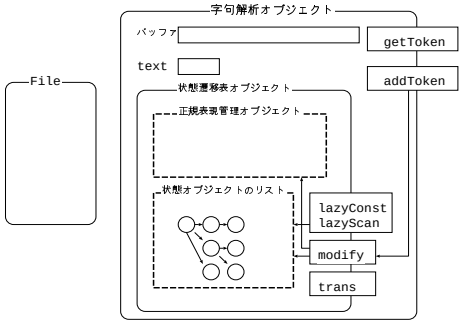

図 4: 字句解析部

以下の手順で字句を解析する．

1. 原始プログラムから 1 文字ずつ文字を読み込み、バッファに保持する． バッファ内の 1 文字を引数として、状態遷移表オブジェクトに遷移のメッセージを送り、トークンとして受理できるかどうかを判断する．
2. もし受理できれば状態遷移表で一致した正規表現のアクションを起こす． このアクションは、あらかじめ定義した Token クラスの派生クラスのインスタン

スを生成することである. 生成されたインスタンスを構文解析部への返り値とする.

以下の手順で字句を拡張する.

1. インタプリタから字句解析オブジェクトに拡張メッセージを送る.
2. 字句解析オブジェクトが拡張メッセージを受け取ると、字句解析オブジェクトが保持する状態遷移表オブジェクトを、先に説明したISGのMODIFYのアルゴリズムを用いて拡張する. 次に新しい字句の正規表現とアクションを保持した **rule** オブジェクトを正規表現管理オブジェクトに登録することで追加された字句の認識が可能となる.

5.2 インタプリタ

y-1 処理系では拡張字句を定義した後にその字句を使用できるようするので、構文を解析しながら逐次実行する方式で、インタプリタを作成する.

原始プログラム中でクラスが定義されたとき、定義されたクラスに対応してクラス classObj のインスタンスが生成される. 生成されたインスタンスは

- クラス名
- メソッドオブジェクトへのポインタ
- 基底クラスへのポインタ
- データメンバオブジェクトへのポインタ

を保持し、記号表オブジェクトに登録される.
処理系が自己表現への拡張メッセージを解釈したとき、追加字句を定義した正規表現と、正規表現と一致したときに起こすアクションを記述したクラスに対応するインスタンスを保持した **rule** オブジェクトを生成する. 生成した **rule** オブジェクトを引数に自己表現 (Self_Representation) に拡張メッセージ addToken を送信する.

6 おわりに

本稿では、我々の提案する字句拡張方式を基づきオブジェクト指向言語 *y-1* の設計を行なった. *y-1* 処理系を試作することにより、字句の拡張が可能であることが確認できた.

現在、IPG(Incremental Parser Generator)[3] を調査し、構文解析部を増殖的に生成する方法を考えている. IPGでは、新構文で用いる文法を文脈自由文法で与えることによって、動的に構文解析部を生成することができる. 増殖的な構文解析部は、本研究で設計・試作した字句解析部と同様な方法で、自己表現にメッセージを送り、構文解析表を再構築することで、実現できると考えている. 増殖的な構文解析部を設計するさいに、各文法記号に属性の集合をどのように結び付けるかが今後の課題である.

謝辞

本研究を進めるに当たり多大な助言を頂き, また、熱心にご指導下さいました南山大学情報管理学科の野呂昌満助教授に深く感謝致します. 野呂昌満助教授には、2年間指導教員として勉強だけでなくあらゆる面で大変お世話になりました. また、有益なアドバイスを頂きました野呂ゼミ卒業生の中原雅仁さん、大学院の塩田康隆さん、山野篤さんに感謝します.

参 考 文 献

[1] Chiba, S: A Metaobject Protocol for C++, in OOPSLA'95 Conference Proceedings Object-Oriented Programming Systems,Languages,and Applications,pp.285-299,1995.

[2] Heering. J., P. Klint and J. Rekers: Incremental Generation of lexical scanners, in ACM Transaction on Programming Languages and Systems, Vol.14,No.4,pp.490-520 1992

[3] Heering. J., P. Klint and J. Rekers: Incremental Generation of Parsers, in IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,VOL .16,NO.12,pp.1344-1351,1990.

[4] Ishikawa, Y: Design and Implementation of Metalevel Architecture in C++ -MPC++ Approach-, inReflection'96 Conference,http://www.rwcp.or.jp/people/ishikawa/mpc++.html, 1996

[5] A.V. エイホ, R. セシイ, J.D. ウルマン: コンパイラ I 原理・技法・ツール サイエンス社

[6] 疋田 輝男, 石畑 清: コンパイラの理論と実現 共立出版

[7] 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.5-14,1994

[8] 中原 雅仁: 構文拡張機能を持つ自己反映オブジェクト指向言語の設計,1996

[9] 一杉 裕志, 松岡 聡, 米澤 明憲: 実行時カーネルのないリフレクティブな並列オブジェクト指向言語の実現方法 コンピュータソフトウェア,Vol.11,No.3,pp.225-237,1994

[10] 増原 英彦, 松岡 聡, 渡部 卓雄: 自己反映並列オブジェクト指向言語 ABCL/R2 の設計と実現 コンピュータソフトウェア,Vol.11,No.3,pp.175-192,1994