第57回 プログラミング･シンポジウム 2016.1.8-10

構文拡張可能なプログラミング言語をいかに設計するか

赤間 仁志[1,a)]　川合 秀実[2,b)]

概要：プログラミング言語の構文を自由に変更できれば，その言語の記述能力をユーザの望むように拡張することができる．しかし既存の言語では構文の拡張を，構文ルールをそのまま書き下すかのように，直感的に行うことは困難である．そこで，本稿ではプログラミング言語の構文を，ユーザが自由に拡張する手法を提案する．本手法の最大の特徴は，特殊な機能を用いず，ごく自然に構文の拡張を記述できることである．そして，この手法を元に設計したプログラミング言語 Garbanzo によって，自然な構文の拡張が可能であることを示す．

キーワード：構文の拡張，メタプログラミング

1. はじめに

プログラミング言語の構文は，それぞれの言語に固有のものであるが，構文を自由に変更できれば，プログラムの記述を簡潔にする構文の導入が可能となり，生産性の向上が期待できる．しかし，既存の言語の多くは構文を拡張する機能を全く有していないか，あるいは限定的な機能しか持たない．

本稿で提案する，構文を自由に拡張する手法を実装したプログラミング言語では，ユーザは直感的に，その言語のプログラムのみを用いて，その言語自身の構文を拡張できる．さらに，構文の拡張を繰り返すことも，構文を拡張する構文を新たに定義することも可能である．

そして，この手法を元に新しく設計したプログラミング言語 Garbanzo を紹介し，この言語で自由な構文の拡張が可能であることを示す．

[1] 東京工業大学 理学部 情報科学科
[2] サイボウズ・ラボ株式会社
[a)] akama.h.ab@m.titech.ac.jp
[b)] h-kawai@labs.cybozu.co.jp

2. 構文を拡張することの利点

C 言語ではプリプロセッサマクロによるテキスト置換により，ユーザが新しい記法を導入できる．例えば，次のプリプロセッサマクロ定義により，指定した回数だけ実行するループ文を新しく定義したかのように見せかけることができる．

```
#define times(i, n) \
    for (int i = 0; i < (n); i++)
```

すると，10 回ループするプログラムは以下のように書ける．

```
times(i, 10) {
    printf("Hello, %d!\n", i);
}
```

新しく導入した `times` 文を用いることで，`for` ループの記述で発生しがちな，インクリメントする変数を間違えるミスの防止や，タイプ量の削減，

および視認性の向上が実現できる．

このように，ユーザが独自の構文を導入することで，プログラミング言語の記法をより洗練させることができる．

また，構文を簡潔にすることで，可読性の向上も期待できる．例えば，Python のようなインデントベースのブロック構文を用いることで，ネスト構造の把握が容易になり，閉じ括弧を用いる必要もなくなる．しかし，そのような構文を C 言語に導入することは，プリプロセッサの機能だけでは不可能である．そのため，C 言語の構文を次のようなソースコードが解釈できるように拡張することはできない．

```
int main():
    char *str = "Hello";
    printf("%s\n", str);
    return 0;
```

この例のように，既存の言語では一般的な構文の拡張を容易に行うことは不可能である．そこで我々は，ユーザが構文を容易に，かつ自由に拡張できるプログラミング言語の設計手法を提案する．

3. 構文拡張の手法と実現方法

本稿では，プログラミング言語の構文の拡張とは，既存の構文ルールの追加，変更，および削除のことを指す．例えば，次のような例がある．

- 新しく，累乗を意味する二項演算子の構文ルールを定義する．
- 先ほど定義した，累乗を意味する構文ルールを式として解釈できるようにする．
- もはや累乗を表す構文ルールは不要だと判断したため，このルールを削除する．

本章では，構文に対するこれらの操作を可能とするプログラミング言語は，どういった要件を満たすべきかを提示し，その要件を満たす構文拡張の方式および，実現のための準備について述べる．

3.1 構文拡張可能なプログラミング言語の要件

構文を拡張可能なプログラミング言語は，以下のような性質を備えることが望ましいと考える．

1. ユーザが容易に構文を拡張できる．
2. 構文ルールを追加することも，変更することも，削除することも可能である．
3. 構文を拡張するための構文も，構文を拡張することで変化させることができる．

1 つ目の条件は，言語処理系の作成に精通していないユーザでも構文を拡張でき，それによって生産性の向上ができるために必要である．

2 つ目の条件が必要であるのは，構文の追加だけでなく，削除および既存の構文の変更も可能であれば，プログラム内の一部分だけの局所的な構文の変更や，不要なルールの削除を実現可能だからである．

3 つ目の条件は，構文を拡張する方法もまたユーザがカスタマイズできることを要求しており，もし構文の拡張方法に不満があった場合にも，柔軟に対処するために必要である．

3.2 構文拡張の方式

構文拡張を行える言語を設計するためには，言語側がユーザに対して，その言語を拡張する手段を提供する必要がある．いま，ある言語 A の構文を拡張可能にしたいとする，するとユーザによる拡張方法として，以下の 3 通りの方式が考えられる．

1. 言語 A の構文を拡張するための言語 B を設計する．
2. 言語 A の構文を拡張するための構文を言語 A に用意する．
3. 言語 A の構文解析器を言語 A のプログラムで記述する．

1 番目は，言語 A とは異なる，構文を拡張するため専用の言語 B を用意する方法である．例えば，構文定義のための特殊なファイルを言語 B で記述することで，言語 A に構文の拡張を提供することが考えられる．そのため，普段言語 A に慣れ親しんでいるユーザが，言語 B を新しく覚える必要がある．また，両方の言語をひとつのソースコードファイルに混在して書くことはできないため，局所的に構文を変化させることはできない．

2 番目は，言語 A に構文を拡張するための構文を用意し，ユーザはその構文を通して言語 A の構

文を拡張する．例えば，C プリプロセッサのマクロがこれに相当する．しかし，この場合は拡張した構文を利用する際に制約が存在し，C プリプロセッサの場合では，マクロ関数の形でしか記述できない．また，構文を拡張する構文を導入することも難しく，拡張性に難がある．

3 番目は，通常は言語処理系に実装される言語 A のソースコードの構文解析器そのものを，言語 A のプログラムとして記述する方法である．そのためこの方式では，処理系の構文解析器が直接的に言語 A のソースコードを抽象構文木に変換するのではなく，処理系は言語 A のプログラムとして書かれた構文解析ルールを実行し，その結果として抽象構文木を生成する．

この手法の利点は，構文解析をされる対象である言語 A のプログラムが，同じ言語の構文解析器を完全に操作することができるため，柔軟性および拡張性に富むことである．さらに，構文拡張の定義と，拡張された構文でのプログラムの記述を混在させることも，構文を拡張する方法をユーザが変更させることも容易である．

そのため，本稿で提案する構文を拡張可能なプログラミング言語の設計手法は，方法 3 をベースとする．

3.3 構文解析器の埋め込み

ある言語の構文解析器を，同じ言語のプログラムで記述する方式を採用すると，その言語の構文解析器に対して，その言語のプログラムからアクセスできる必要がある．そのため，処理系はその言語のプログラムに対して，構文解析器へのインターフェイスを提供する．

本方式では，構文解析器を変更する方法として，処理系に構文解析を行う命令や関数を実装したのち，構文解析ルールを同じ言語のプログラムとして記述し，それらを一つ一つのルールとして構文解析器に登録するといった方針をとる．

3.4 構文解析ルールの表現

構文解析ルールを記述する方法として，再帰下降パーサが知られている [2]．再帰下降パーサには，構文解析ルールをそのままプログラムとして書き起こせるという利点がある．そのため，ユーザは構文解析ルールを他のプログラムと同じように記述することができる．

また，ユーザが構文解析ルールを動的に変更可能とするためには，構文ルールをいくつかの小さいルールの組み合わせとして記述し，その一部を後から変更可能にすることで対処する．いくつかの小さな再帰下降パーサを組み合わせて目的のパーサを構成する実用的な技法として，パーサコンビネータ [4] が知られている．

本手法では，一つの構文ルールを，複数の小さな構文ルールを組み合わせた状態で保持することで，あとから構文ルールを変更することを可能としている．

3.5 構文解析の流れ

まず，本手法により設計するプログラミング言語 A の処理系には，構文解析器を操作するための最低限の命令と，基本的なデータ型の構文解析に必要な構文解析ルールを，あらかじめ搭載しておく必要がある．

そして実際の実行では言語 A の処理系から，言語 A で記述された構文解析器のコードを呼び出し，得られた構文解析結果のプログラムや関数定義を処理系が評価する．そのため，構文解析器を変更する命令を評価すれば，構文解析の次のステップから新しく導入した構文でソースコードを記述できる．

この方式を実装する言語の処理系がコンパイラであるか，インタプリタであるかに関わらず，プログラムは一度，構文解析のタイミングで評価される．そして，そのプログラムの評価に伴い，構文解析器に対してルールの追加や変更および削除を行うことで，その次に実行される構文解析の動作を変更させる．

これらの構文解析器に対する，ルールの追加・変更・削除といった操作は，データ構造のフィールドに対する代入操作のように行われる．これらの操作は構文の拡張とは関係のない普通のプログラムでよく行われる操作であり，命令的なプログ

ラミングの経験がある人にとっては慣れ親しんだものである．そのため，構文の拡張という特殊な操作を，データ構造の操作のように自然に扱うことができる．

4. Garbanzo 言語

本稿で提案するプログラミング言語 Garbanzo は，前章で述べた構文拡張手法を持つシンプルなプログラミング言語である．この言語は動的型付けであり，処理系はインタプリタ方式で動作する．また，この言語は設計段階であり，プロトタイプを Ruby で実装している．

4.1 データ型

Garbanzo には，文字列，数値，真偽値などの基本型と，関数，およびデータストアと呼ばれるデータ型が存在する．データストアは順序を持つ連想配列であり，例えば以下のようなものである．

```
{ melon: 42, pumpkin: true }
```

このデータストアは，Garbanzo のプログラムの抽象構文木の記述や，変数を格納する環境としても用いる．

4.2 文の評価

Garbanzo のインタプリタは，データストアを評価した時に，その要素の"@"（アットマーク）というキーに対応する値を組み込みの命令名の文字列と解釈し，残りの要素をその命令の引数として命令を発行する．組み込みの命令には，データストアへの代入や要素の取得，算術演算や論理演算，逐次実行など制御構造に関わる命令などが存在する．

構文解析を行う上で特に重要な命令として，choice 命令がある．choice 命令は引数として与えられた，いくつかの命令を順次実行し，最初に成功した命令の結果を返却する命令である．そのため，choice 命令によって，A または B といったルールを記述できる．

構文ルールの変更は，構文解析に関する命令を含んだプログラムを直接書き換え，これらの命令の引数部分にルールを追加，変更または削除することで行う．さらに choice 命令の引数は順序の情報を持っているため，ユーザが新しく作成する構文ルールの優先順位も指定して挿入することもできる．

このように，Garbanzo はプログラムを書き換えるスタイルのプログラミングをサポートする．

4.3 構文解析ルール

Garbanzo において，構文解析ルールを表すプログラムは，source という名前の変数に格納された文字列を対象に構文解析を行い，成功したならば source の内容を変更し，結果としてプログラムの構文木を構築して，失敗したならばエラーを返すプログラムである．いわゆる，破壊的な操作を行うシンプルな再帰下降パーサであると言える．

Garbanzo の構文解析ルールは，parser という変数名のデータストアに名前をつけて格納する．例えば，文字列の表現（"string" など）を読み取るルールであれば，parser の中の，string という場所に格納する．

4.4 実行の流れ

Garbanzo のソースコードの構文解析を行うのは，Garbanzo のプログラムである．そのためインタプリタは，Garbanzo のプログラムを次の方法で実行する．

1. 入力ソースコードを source という変数に文字列として格納する．
2. 以下の手順を，source が空になるまで繰り返す．
    1. 特別なルール sentence を実行することで構文解析を行う．
    2. 得られた結果のプログラムを評価することで，プログラムを一文だけ実行する．

この動作方法により，Garbanzo のプログラムが入力のソースコードを，直接構文解析することが可能となる．

4.5 Garbanzo の初期構文

Garbanzo のプログラムを記述するためには，データストアおよび文字列の構文解析ルールが最

低限必要であるため，インタプリタの埋め込みの構文ルールとして実装する．しかし，これらの構文のみでは冗長になるため，基本的な命令を少ない記述量で発行するための構文を Garbanzo のプログラムとして記述し，これらを合わせた構文を，Garbanzo の初期構文とする．

5. 構文拡張の例

Garbanzo での構文拡張の例として，Garbanzo の構文に C 言語のような後置インクリメントを導入することを考える．すなわち，a++と記述すると，a = a + 1 と解釈されるように Garbanzo の構文解析ルールを拡張する．

リスト 1 のコードで，Garbanzo の構文解析器を拡張し，後置インクリメント構文を導入できる．

```
parser.increment = do                    # (1)
    name = $parser.symbol                # (2)
    terminal { string: "++" }            # (3)
    ‘%(name) = %(name) + 1‘              # (4)
end

parser.sentence.children.increment =
    parser.increment                     # (5)
```

リスト 1. 後置インクリメント 1

この構文拡張の例では，まず新しく increment というルールを定義してから，この increment というルールを文のひとつとして解釈できるよう，文を意味するルール sentence に追加している．

リスト 1 は次のように解釈される．まず，do から end で囲まれた部分は，複数の文 (sentence) を順番に実行するという命令をクオートするという構文であり，これは実行可能なプログラムの断片と解釈される．そのため，最初の代入文 (1) では，parser.increment に，構文解析ルールを表現したプログラムを格納している．

この do〜end の内部がルールとして実行されると，まず (2) の文が実行され局所変数 name に，$parser.symbol の結果を代入する．ここで，$は式を評価 (eval) する前置演算子であり，これはシンボルを意味する構文解析ルールを実行することを表している，次に (3) では，指定した文字列を 1 つ読み込む組み込み命令 terminal を，引数名 string に"++"を与えて実行する．これらの行 (2) および (3) によって，後置インクリメントの構文解析が行われる．do〜end の最後に，(4) ではこのルールの構文解析結果として，準クオート [1] を用いて代入文を構成している．準クオートの内部は，クオートされるが， その中の %(name) によって，name の部分がアンクオートされ，(2) で読み込んだシンボルがここに埋め込まれる．

そして，先ほど定義した increment ルールを文の一つとして認識されるためには，sentence というルールに追加する必要がある．sentence ルールは，複数の候補を順番に試す choice 命令である．この候補は，sentence.children という箇所に列挙されるため，(5) では，その中へ increment という名前をつけて，ルールを追加している．

Garbanzo では，構文を拡張することは，プログラムを意味するデータストアを変更ことに他ならない．そのため，代入文を生成する構文を新たに定義することで，構文を拡張する構文もまた同様に拡張できる．

リスト 1 の (5) の，構文解析ルールの追加の記述はやや冗長である．そのため，ルールを追加するルールを定義することを考える．リスト 2 では，choice 命令に対して，新しくルールを追加する代入演算子 addrule（|=）を導入している．

```
parser.addrule = do
    target = $parser.path
    terminal { string: "|=" }
    name = $parser.symbol
    ‘%(target).children.%(name) =
        parser.%(name)‘
end

parser.sentence.children.addrule =
    parser.addrule
```

リスト 2. 構文拡張の拡張

この addrule 演算子を用いると，リスト 1 の (5) で示した，ルールを追加する部分が，リスト 3 の

ように書ける．

```
parser.sentence |= increment
```

リスト 3．後置インクリメント 2

このように，本稿で提案する手法では，構文を拡張するためだけの専用の構文を用いないため，プログラミング言語の構文の拡張を繰り返し行うことや，構文の拡張方法の拡張も可能とする．これによって，より簡潔な表記を，ユーザが望むままに導入できる．

6. 今後の展望

本稿の手法を用いることで，プログラミング言語の構文をそれ自身のプログラムで拡張できることを示し，その具体例として Garbanzo 言語を提案した．しかし構文の拡張は強力である反面，扱いを誤るとユーザの意図しない動作を引き起こしやすい．これらの問題に対しても，将来的には構文拡張を用いることで対処することが可能であると考えている．

6.1 左再帰の除去

ある言語の構文解析器を再帰下降パーサとして，同一の言語のプログラムとして記述する方法は，一見シンプルで強力だが，再帰下降パーサの弱点をそのまま持つことになる．その一つに，左再帰を持つルールを直接記述すると，無限ループに陥ってしまうことがある．しかし，左再帰を含むルールに実用的なものは比較的多く，例えば，加法は左結合的な二項演算子であるため，BNF で以下のように表される．

```
EXPR ::= EXPR '+' NUMBER
```

そのため，現状の Garbanzo 言語に足し算のルールを直接追加することは困難である．そこで，左再帰を含むルールを記述する専用の構文を導入し，左再帰を除去することにより，この問題を解決する予定である．

6.2 構文のユニットテスト

構文の拡張は一種のメタプログラミングであり，非常に強力な機能である．それゆえ，構文の拡張の際にバグを埋め込む可能性は高く，望んだ構文を得られないことや，入力ソースコードのコーナーケースにて意図しない挙動を示す可能性が否定できない．

この問題の解決には，ユニットテストを行う構文を定義し，ソースコードから望みのプログラムが出力できるかをテストすることを想定している．

6.3 構文木の構築

本稿の手法では，構文解析ルールを表すプログラムが，構文解析結果のプログラムを構築する．そのため，どの式を構文解析時に評価し，どの式を評価時に実行するかを細かく制御する必要性がある．Lisp 系の諸言語では，この問題を準クオートという方法で対処している [1] ．Lisp では S 式を読み取るルールのみを考えれば良いため，アンクオートを 1 箇所で定義すればでよい．しかし，Garbanzo では複数のルールが存在するため，文字列を読み取るルール，式を読み取るルールなど，一つ一つにアンクオートを定義する必要がある．

この問題に対しては，新しくルールを作成する際，自動的にアンクオートを定義する構文を作成することで，アンクオートの定義をその都度行う必要がなくなると考えられる．

7. 既存の手法

7.1 C プリプロセッサ

C のプリプロセッサは，マクロを関数呼び出しの形式で呼び出すことで，単純なテキスト置換を行うものであり，多くのプログラムで用いられている [3]．しかし C のプリプロセッサは単純なテキスト置換であり，それ以上の複雑なことは不可能である．

7.2 Common Lisp

Common Lisp [5] には，リードマクロと呼ばれる構文を拡張できる強力な機能が存在する．この機能を用いると，S 式の構文解析器に，ある一文字に対する構文解析ルールを関数として登録することで，S 式の記法を拡張できる．

Common Lisp のリードマクロと，Garbanzo の構文拡張の大きな違いは，リードマクロでは構文解析を開始する先頭文字を指定してリードテーブルを変更するため，例えば後置インクリメントのような構文を導入することができないことである．その点，Garbanzo では構文解析ルールをそのまま表現した再帰下降パーサにより構文解析を行うため，ユーザは直感的に，複雑な構文を導入できる．

8. 結論

本稿では，プログラミング言語の構文をユーザが自由に拡張するための手法を提案し，その手法を元に作成したプログラミング言語 Garbanzo にて，実際に構文の拡張を柔軟に行えることを示した．

はじめにプログラミング言語の構文をユーザが拡張できることの利点を述べた．構文を拡張することにより，単に見た目を改善できるだけでなく，簡潔な構文を導入することでの生産性の向上が期待できる．

次に，プログラミング言語の構文をいかに拡張するかを論じた．本稿では，ある言語 A の構文を拡張する機能は同じ言語 A のプログラムで制御できることが，柔軟性および拡張性の点から最も望ましいと述べた．

そして，本手法を実装するために必要な機能として，構文解析器の構成方法を具体的に述べた．

最後に，本手法を実装した言語 Garbanzo を設計および実装し，自由な構文の拡張が可能であることを示した．

Garbanzo 言語では，構文解析ルールを Garbanzo 言語そのもので記述することで，構文の拡張，および構文を拡張する方法も拡張できる．

謝辞 本研究はサイボウズ・ラボユースの支援の元で行われている．サイボウズラボ・ユースの積極的な支援にこの場を借りて感謝する．

参考文献

[1] Bawden, A.: Quasiquotation in Lisp, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pp. 4–12 (1999).

[2] Burge, W.: *Recursive Programming Techniques*, Addison-Wesley (1975).

[3] Ernst, M. D., Badros, G. J. and Notkin, D.: An empirical analysis of C preprocessor use, *Software Engineering, IEEE Transactions on*, Vol. 28, No. 12, pp. 1146–1170 (2002).

[4] Hutton, G. and Meijer, E.: Monadic Parsing in Haskell, *Journal of Functional Programming*, Vol. 8, No. 4, pp. 437–444 (online), DOI: 10.1017/S0956796898003050 (1998).

[5] Steele, G. L.: *Common LISP: the language*, Digital press (1990).