ヒューマンインタフェース 95－17
コンピュータビジョンと イメージメディア 129－17
(2001. 9. 14)

# 言語編集過程の統合支援フレームワークEPO

広瀬雄二* 大駒誠一†
東北公益文科大学

2001年9月14日

**概要**

われわれが、考えていることを計算機に伝えるとき、計算機用に設計された言語を利用する。計算機に仕事をさせたい場合はプログラミング言語を利用するし、テキストを構造を保ったまま書き記したい場合はマークアップ言語を利用する。それらを入力する時に利用するテキストエディタには、利用言語に応じてそれらの記述労力を軽減するさまざまな支援ツールが存在する。しかしそれらのツールは言語により全く独立したソフトウェアとして提供されているのが現状である。本稿では、計算機言語の入力全般に共通した編集作業工程を抽出し、あらゆる計算機言語の入力工程を支援するパッケージを、対象言語に依存した情報を与えるだけで作成できる統合支援環境フレームワーク“EPO”を提案する。

# EPO - A Framework of Integrated Supporting Environment for Language Editing Process

HIROSE Yuuji
OKOMA Seiichi
Tohoku University of Community Service and Science

**Abstract**

We use language designed for computers to express and transmit our notion to computers. We use programming language to make computers do some job, while we use mark-up language to describe our thought with its logical structure. There are many kinds of tools on text editor to support various editing process according to computer languages. However, these tools are designed for only one language, consequently we can't apply one supporting tool to other languages. In this paper, we try to extract common editing process among all computer languages and construct supporting methods for those common processes into a framework. "EPO" is a framework of integrated supporting environment for language-editing, by which we can produce a supporting environment through only giving language dependent parameters to EPO.

*yuuji@koeki-u.ac.jp
†okoma@ae.keio.ac.jp

# 1 背景

人間の考えを計算機に伝える場合には計算機言語を利用する。人間同士のコミュニケーションでは、自然言語を用いて、思いついたことを曖昧に並べるだけでもニュアンスは伝わるのだが、計算機相手ではそのようにはいかない。このため、計算機言語は文法も明確に定義され、伝えんとする内容の構造を正確に反映するための多数の構文が用意されている。

たとえば、広く普及しているプログラミング言語のCでは、単に文を書き連ねると、それぞれの文は書いた順に実行されていく。その一部のかたまりを、ある条件が満たされる間繰り返されるような構造を記述する場合は、

```
while (条件) {
  ...
}
```

のように、「while 構文」を利用して、繰り返し構造が適用される範囲を明確にする。いっぽう、マークアップ言語 LaTeX[1] では、文書中に「箇条書き」として列挙する項目を記述したい場合は

```
\begin{itemize}
 \item ...
 \item ...
\end{itemize}
```

のような構文を利用することで、文書中の他の部分との構造の違いを明確化できる。

プログラミング言語、マークアップ言語、いずれの場合も構造を決定する構文は利用言語によって明確に定義されているため、それらをテキスト中に記述する際には正しい構文を書き下す必要がある。LaTeX や HTML などの代表的なマークアップ言語の場合は、このための構文が短くないこともあり、それらを手入力する時に無視できない負担が発生している。

また、特定の計算機言語で書かれたテキストは通常、「コンパイル」や「タイプセット」などの操作を行なって、計算機が直接解釈できる形式 (あるいはより直接的に近い形式) に変換して「成果物」として利用する。この過程で、変換処理時に文法的エラーなどが言語処理系から報告され、人間はこれを直して再度変換処理を行なう、といった作業を繰り返す。このような作業は、単純なものではあるが何度も繰り返す性質のものであるため累積負荷は大きなものとなる。

このような、計算機言語を利用する場合の定型的作業を軽減する目的で多数の「入力支援環境」が提供されてきた。ここでは、Unix 系のシステムで開発され、最近では Windows 系列、MacOS 系列のシステムにも移植されて利用されているテキストエディタ GNU Emacs [2] を対象として、その上で動作する各支援環境を考察する。

# 2 既存の言語入力支援環境

代表的な言語入力支援環境として、C 言語プログラミングを支援する CC Mode[3] と、マークアップ言語利用を支援する野鳥 [4] をとりあげ、それらの支援内容を分析する。

1. CC Mode

   "CC Mode" は GNU Emacs の配布初期の頃から標準付属パッケージとして添付されていた c-mode.el を元にさまざまな改良が加えられたもので、現在は Martin Stjernholm らが保守している C, C++, Objective-C, Java 言語の入力支援を行なうメジャーモードである。CC Mode に備わる主な支援機能には、

   - プログラマの好みのスタイルに応じたソースプログラムのインデント
   - コメント文の自動行分割
   - 関数や文を単位としたポイント移動

   などがある。基本的に C 言語の文法を考慮して、論理構造に応じたソースの整形を主眼としたパッケージである。

2. 野鳥

「野鳥」は筆者により開発されたもので、LaTeX を利用したソーステキスト作成にかかわる作業を軽減し、入力の効率化を促進する支援環境である。野鳥に備わる主な支援機能は以下のようになっている。

- LaTeX 関連外部プロセスの起動
- エラー箇所への自動ジャンプ
- 各種 LaTeX マクロの補完入力 (&学習)
- 既入力したテキストの括り補完
- セクション区切り入力時の文書構造アウトライン表示
- セクションコマンドの一括シフト
- 既入力 LaTeX マクロの削除/変更
- ファイル間、begin/end 間、ref/label 間 cite/bibitem 間ジャンプ
- 一括コメントアウト/アンコメントアウト
- アクセント記号/数式環境マクロ/ギリシャ文字の入力支援
- 標準的 LaTeX マクロのオンラインヘルプ
- ドキュメントのインクルード構造の視覚的表示

上記に代表される野鳥の持つ編集操作支援体系は、多くの利用者に支持された。それを表すことがらの一つとして、野鳥と同じ支援機能をもつソフトウェアが GNU Emacs 以外のテキストエディタに移植実装されたことが挙げられる (表 1)。

表 1: 野鳥アーキテクチャの移植実装

| OS | エディタ | 実装名 |
|---|---|---|
| MS-DOS | Vz | 雷鳥/LaiTeX (桂川直己氏) |
| Windows | Wz | 白鳥/HackTeX (竹中浩氏) |
| Windows | 秀丸 | 飛鳥/HiTeX (安田晴行氏) |
| Windows | xyzzy | 花鳥/KaTeX (前田学氏) |

## 3 既存の支援環境の問題点

多種多様の便利な支援環境が存在し、多くの計算機言語の入力負担が軽減されている現状であるが、これは同時に以下のような問題点をも意味する。

- マイナー言語への対応の弱さ

  C や LaTeX などは利用者数が多いので、支援環境も用意されやすい。しかし、利用者数の少ない言語や、新たに登場した言語の場合は、その支援環境が用意されることは期待しにくい。自分で作ることは多くの人にとって容易でない。

- インタフェースの非統一性

  たとえば「言語処理系を呼ぶ」操作一つをとっても支援環境が異なれば操作方法も異なる。類似した操作が同じキー割り当てになっているとは限らないので、複数の支援環境を使う場合にはそれぞれの操作体系を独立して覚える必要がある。

これらの問題を解決するためには、プログラミング言語/マークアップ言語それぞれに共通な編集工程を抽出し、それらの操作を言語によらず同一の操作体系で補佐する統合的な支援環境を構築することが有効であると予想される。

## 4 言語入力作業の共通項

プログラミング言語用の支援環境とマークアップ言語用の支援環境を俯瞰すると、いずれにも共通した編集工程があることが分かる。

### 4.1 箱構造の入力

C などのプログラミング言語や、LaTeX などのマークアップ言語いずれの場合も、テキストの一部を他の部分と異なった論理構造であることを示すための記法を他数配置することにな

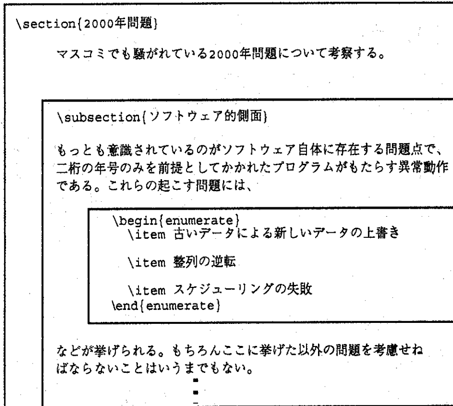


図 1: LaTeX 入力ソースの箱表現

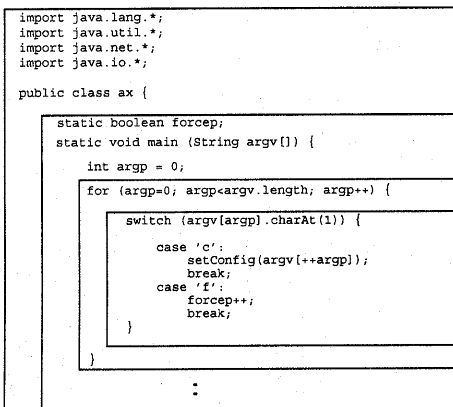


図 2: Java ソースの箱表現

る。たとえば、LaTeX を対象言語とするソーステキストの場合、図 1 に示すような箱構造を列挙したものと捉えることができる。この類推をプログラミング言語に適用すると、たとえば Java 言語によるプログラムを論理構造のかたまりと捉えた場合、図 2 のような入れ子状の箱の配列と考えることができる。プログラミング言語、マークアップ言語、いずれの場合も構造の範囲を限定するキーワードは言語固有の、明確に定められた記法となっている。以後、言語固有の記法によって前後範囲を定められたテキスト中の特定の部分をたんに「箱」と呼ぶことにする。

### 4.2 連関操作

ソーステキスト中で構造の単位をなす「箱」は互いに無関係に存在しているわけではない。たとえば、プログラミング言語において定義される「関数」は上位レベルの「箱」と捉えられるが、これは単独で位置することはまれで、プログラムのどこか別の部分から参照されることを前提として作成されるものである。同様に、マークアップ言語においても、文書の特定の部分に「ラベル」をつけ他所から参照することが可能である。文書中の任意の二点間を一つの識別子を用いて結びつける作業も計算機言語に共通したものと考えられる。

多くの支援環境では、なんらかの参照を意味する単語をテキスト中に入力する場合、その識別子を補完入力によって行なうことが可能である。これは入力時のタイプ数を軽減するだけでなく、識別子の綴り誤りを回避しソーステキストの文法的エラーの発生確率を下げる働きも持っている。

ソーステキスト中で定義と参照を作成したり消去修正したりする操作のことを以後「連関操作」と呼ぶことにする。

### 4.3 言語処理系起動操作

プログラミング言語ではコンパイラ、マークアップ言語ではタイプセッタや文法チェッカを起動する処理が行なわれる。これらの言語処理系の起動はソーステキストを保存したファイルに対して行なわれるという点でいずれの言語においても共通の操作だといえる。

## 5 統合支援フレームワーク EPO

上述の問題点と編集操作工程の性質をふまえて、対象言語に依存しない編集操作を支援するフレームワーク EPO (The Editing Process Organizer) を設計・実装した。

## 5.1 EPOの概要

EPOでは、典型的な言語編集操作群を大別して以下の4つに分類している。

1. 構造の新規入力操作

   特定の「箱」を形成する言語固有の記法を入力する操作。たとえば LaTeX で \begin{itemize}〜\end{itemize}を入力する操作がこれに当たる。

2. 既入力構造の修正

   既入力テキスト中に存在する言語固有の記法を別の記法に変更する操作。たとえば LaTeX で\begin{itemize}〜\end{itemize} を\begin{enumerate}〜\end{enumerate} に変更する操作がこれに当たる。

3. 連関操作

   ソーステキスト中の任意の二点を、識別子を利用して結びつける操作。プログラミング言語では関数の参照と定義、マークアップ言語ではクロスリファレンスやハイパーリンクの定義と参照部分の作成がこれに当たる。

4. 言語処理系の起動操作

   ソーステキストを(保持するファイルを与えて)文法的に解析したり、別のオブジェクトに変換したり、インタプリタ実行したりする操作全般がこれに当たる。

EPOは、これら四工程に内在する言語非依存性質を抽出して得られた、編集操作支援の「方法」だけを知識として持っている。これに対象言語に依存した文法的な情報や、言語処理系の起動方法にかかわる知識を与えることで、その言語用の支援環境として機能するようになる(図3)。四工程に対応したモジュールは、EPOI[1], EPOC[2], EPOR[3], EPOP[4] という名前

[1]EPO Input Aider
[2]EPO Change Operation Supporter
[3]EPO Relation Resolver
[4]EPO Process Handler

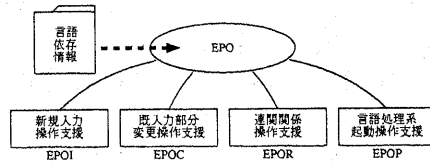


図3: EPOの構成概念図

となっている。

EPOの実装は、GNU Emacs エディタに内蔵された Emacs-Lisp 言語で行なった[5]。

## 5.2 EPOの基本支援機能

EPOで支援可能な言語編集工程は以下の通りである(括弧内は該当するモジュール)。

- 構文新規入力時の補完入力 (EPOI)
- 反復指定子の自動生成 (EPOI)
- 構文修正作業 (EPOC)
- 連関の作成・追跡・変更・抹消 (EPOR)
- 識別子の随時補完 (EPOR)
- 言語関連外部プロセス起動処理 (EPOP)
  - コンパイル/タイプセット処理過程
  - プログラムの試走過程 / テキストのプレビュー過程
  - 被参照過程 (java → html → appletviewer など)

それぞれについて動作例を加えて解説する。

### 5.2.1 構文新規入力時の補完入力

各種言語で論理的なひとかたまりの「箱」を形成する構文は、その構文の種類を示すキー

[5]GNU Emacs 19.28, 20.0, 21.0(pre) および XEmacs 21 での動作を確認している。

ワードのみを補完入力することで、形成構文全体を入力することができる。たとえばHTMLを利用する場合、Un-ordered List エレメントを形成するには、<ul>〜</ul>両タグを使用する。EPO の補完機能でこれを入力する場合まず、構文入力補完用のキーバインド C-c C-s をタイプする。ミニバッファに、構文補完メニューが現れる。

```
-かんな-EJJ:---F1  hoge.html        [1]    (yahtml EPO E
Completion : (b)block (i)inline:
```

この例では、EPO に HTML 言語の構文として「ブロックタイプ (b)」と「インラインタイプ (i)」が登録されているので、二者択一メニューが出される。ブロックタイプ (b) を選択する。すると、キーワードの入力を求めるので

```
m-----------EJJ:---F1  hoge.html        [1]    (yahtml EPO E
Block: ul
```

エレメント名 "ul" を入力する。これにより、テキストに

```
<ul>
</ul>
```

が挿入される。

### 5.2.2 反復指定子の自動生成

言語によらず、ある特定の「箱」の中にその箱固有の要素が繰り返し現れることがある。たとえば、C などのプログラミング言語では、「switch 構文」の中には、"case" が繰り返し登場することが多く、LaTeX の「itemize 環境」の中には "\item" が繰り返し登場する。このように文脈によって決定する繰り返し要素のことを以後、**反復指定子**と呼ぶ。既出の HTML での、<ul>〜</ul>両タグに囲まれたポイント位置で、EPO の反復指定子自動挿入用のキーバインド C-c C-i をタイプする。すると、ポイント位置が ul エレメント箱の内部であることを判別し、テキストに<li>が自動挿入され、結果として

```
<ul>
 <li>
</ul>
```

という内容になる。

### 5.2.3 構文修正作業

手書き文書よりも、テキストエディタを用いるときの利点として、試行錯誤的作業が手軽にできるという点がある。LaTeX を例にすると、itemize 環境で項目列挙した後、それらを enumerate 環境に変える、といった作業も試行錯誤的作業の一つで、EPO ではそのための編集工程を補佐できる。

```
\begin{itemize}
 ...
\end{itemize}
```

のようなテキストで\begin{itemize}または\end{itemize}句のいずれかにポイントを合わせて、構文修正用キーバインド C-c C-c をタイプする。"itemize" を何に変更するか聞いて来るので、

```
m-----------EJS:**-F1  hoge.tex        [1]    (やてふ EPO E
Change `itemize' to: enumerate
```

"enumerate" と入力 (補完可能) すると、

```
\begin{enumerate}
 ...
\end{enumerate}
```

と二箇所に存在するキーワードが同時に置換される。

### 5.2.4 連関の作成・追跡・変更・抹消

ソーステキスト中に記述する識別子が、別箇所に存在する特定の資源を指し示す結果となる場合、それを作成・追跡・変更・抹消する編集工程を補佐する。EPO が認識する連関には、

**参照** 関数またはラベルの定義/参照

**ファイル指示** 同一ファイルシステムに存在する別ファイルの参照を意味する構文

ハイパーリンク 外部資源のリンク指定

**「箱」の境界指定構文** 「箱」の開始位置と終端位置を指定するその言語固有の構文そのもの同士

がある。連関の端点を意味する構文位置にポイントを置いて、連関追跡用のキーバインド C-c C-r をタイプすると、連関の対となるものが編集可能なファイルに存在するものであればそこにポイント移動し、そうでなければ対をブラウザ表示する。もし、ラベルや関数の参照を意味する構文で C-c C-r をタイプしたものの、その定義部分が存在しなかった場合にはこれから定義するものとみなし、定義のための構文テンプレートが生成される。

また、連関削除用のキーバインド C-c C-k をタイプすると、連関の両端が削除される。

#### 5.2.5 識別子の随時補完

ソーステキスト中で識別子を入力する際、C-c C-@をタイプすると、別箇所で定義されている識別子群を候補とした入力補完が行なわれる。

#### 5.2.6 言語関連外部プロセス起動処理

ソースファイルを言語処理系にかける処理など、外部プログラムを起動して行なう操作すべてが、EPO の支援対象となっている。C-c C-t をタイプすると、登録されているその言語固有の関連プログラム起動の一覧が現れる。たとえば、C 言語の場合

```
-かんな-EJJ:---F1  hoge.c          [1]    (C EPO Ekb Ab
Start process: (j)Compile (r)run-it:
```

のように、「コンパイル (j)」と「実走 (r)」の二者択一メニューが出る。ここで j をタイプすると、あらかじめ登録されたコンパイルに必要な外部プロセスが起動される。

### 5.3 EPO に与える言語依存情報

EPO に対象言語の編集支援を行なわせるためには、4 つのモジュールが具体的な動作を決定するのに必要な言語依存情報を与える必要がある。

1. EPOI(構造の新規入力支援) に与える情報

   言語固有の構文を形式ごとに分類して、それぞれに対し、分類名、構文入力時の流れ、キーワードの位置、構文に与える引数の位置と与える形式、キーワードの補完候補などを持たせた情報を与える。

   また、反復指定子に関しても、その反復指定子が存在しうる「箱」を検索するためのパターンとともに指定子要素入力時の流れを与える。

2. EPOC(既入力構造の変更支援) に与える情報

   EPOC は EPOI 用の情報と、後述する EPOR 用の情報を利用して支援操作を行なう。

3. EPOR(連関操作支援) に与える情報

   対象言語に存在する連関を種類ごとに分類して、それぞれに対し、分類名、連関の種類[6]、識別子として利用可能なパターン、などを持たせた情報を与える。

4. EPOP(外部プロセス起動支援) に与える情報

   対象言語利用時に呼び出す外部プロセスを種類ごとに分類して、それぞれに対し、分類名、コマンドライン生成方法などを持たせた情報を与える。

これらの情報は、Lisp 言語のデータ表現に用いる **S 式**で表現して与える。図 4 に実際の S 式表現の例を示した。

[6] 5.2.4 節で挙げたもののうちのどれか

```
(defvar epo-html-structure-alist
  '((?b (type . block)
        (structure "<" keyword argument ">\n" indent
                   cursor "\n" "</" keyword ">" indent)
        (table . epo-html-block-table)
        (argsep . sgml)
        (mustmatch . t))
    (?i (type . inline)
        (structure "<" keyword argument ">" cursor
                   "</" keyword ">" indent)
        (table . epo-html-inline-table)
        (argsep " " "" "=" " " "" "")
        (arg-reader . nil))))
```

図 4: EPO に与える言語依存情報例 (抜粋)

表 2: 支援可能化した言語一覧

| | |
|---|---|
| プログラミング言語 | C, Java, Perl, Ruby, Emacs-Lisp, CORBA/IDL, Xi |
| マークアップ言語 | LaTeX, HTML |
| 定義用言語 | RELAX, Ant |

## 6 評価

EPO に実際に言語依存情報を与えて支援機能が提供させる試行を行なった言語一覧を表 2 に掲げる。

いずれの言語においても 5.2 節で列挙した編集工程を、同一インタフェースで支援できるようになった。また、各言語の言語依存情報を記述したファイルの大きさ[7]は表 3 のようになっており、個人レベルで十分に作成できる大きさのものといえる。このことは、新しい言語やマイナーな言語を利用する場合にも、その言語によるテキストの記述を支援する環境を容易に得られることを意味する。

表 3: 言語依存情報記述ファイルの大きさ

| 言語 | 行数 | バイト数 |
|---|---|---|
| Ant | 82 | 2332 |
| C | 134 | 4093 |
| CORBA/IDL | 117 | 3471 |
| Emacs-Lisp | 105 | 3325 |
| HTML | 155 | 4879 |
| Java | 122 | 5019 |
| LaTeX | 138 | 4280 |
| Perl | 107 | 4278 |
| RELAX | 91 | 2690 |
| Ruby | 91 | 2706 |
| Xi | 89 | 2451 |

## 7 結論

計算機言語用のソーステキストの編集作業には、言語に依存しないいくつかの典型的工程が存在する。それらの工程の「手続きルール」だけを抽出して支援方法を実装したフレームワークを作成し、それに対象言語に依存した情報を与えて実際の支援環境を作成する手法を取ることで、より多くの計算機言語編集用の支援環境を小さなコストで構築できることが分かった。

[7]コメント行等を除く

## 参考文献


[1] Leslie Lamport; LaTeX: A Document Preparation System; Addison-Wesley, Reading, Mass. (1986).

[2] The Free Software Foundation; GNU emacs - Extensible, real-time editor; http://www.gnu.org/directory/emacs.html

[3] The Free Software Foundation: 2000- Martin Stjernholm, 1998-1999 Barry A. Warsaw and Martin Stjernholm, 1992-1997 Barry A. Warsaw, 1987 Dave Detlefs and Stewart Clamen, 1985 Richard M. Stallman; cc-mode.el — major mode for editing C, C++, Objective-C, and Java code; http://cc-mode.sourceforge.net/

[4] 1991- HIROSE, Yuuji; 野鳥 - Yet Another tex-mode for emacs; http://www.yatex.org/