多相型言語の変数名補完を行うEmacsモードの開発

後藤 拓実, 篠埜 功

芝浦工業大学 工学部 情報工学科

{l06050,sasano}@sic.shibaura-it.ac.jp

概 要 変数名補完はEclipseやVisual Studio等の開発環境で広く利用されている機能である。補完の候補はカーソル位置において有効な変数であり、変数の型情報により絞り込みを行うと、コンパイル時の型エラーを減らす効果がある。変数の型の記述を省略出来るStandard MLやHaskell等の言語においては変数の型を得るために型推論を行う必要があるが、作成途中のソースコードを用いるため不完全な情報で型推論を行わなければならない。本論文ではカーソル以前のプログラムが完全に与えられている場合を対象として、型推論機構を備えた多相型言語について変数名補完を行う手法を提案する。提案手法の有用性を示すため、簡易言語に対する変数名補完をEmacsモードとして実装した。

1 はじめに

変数名補完はEclipseやVisual Studio等の統合開発環境に実装され広く利用されているプログラミング支援機能であり、プログラマがソースコードを入力する際にカーソル位置にて有効な変数の一覧を表示し、変数名を選択するとその変数名を補うという機能である。変数名補完により、プログラマの入力の負担が軽減され、スペルミスが減少するため、variable not foundのようなコンパイル時エラーが抑制される効果がある。また、通常ある程度大きなプログラムではソースコードの可読性を上げるために長い変数名を用いるが、変数名の補完機能により、長い変数名でも効率良く入力することができる。

変数名補完で型情報を考慮して絞り込みを行うと、スペルミスだけでなく型エラーも減少させる効果が期待できる。本研究では、型推論機構を持つ静的型付き言語を対象とし、型情報を考慮した変数名補完を行う方式を提案する。変数の型を推論するためには構文解析を行う必要があるが、編集中のソースコードは不完全なため、部分的な構文木しか得られない。この問題に対処するため、現在のカーソル位置までのソースコードが完全に記述されていることを前提とし、それ以降の構文木をダミーの構文木で補うことによる補完候補計算方式を提示する。

対象言語の文法はLR(1)とし、カーソル位置までを構文解析した時点でのスタックの情報とその時点までに行った還元の情報から構文木を作成する。対象言語の型システムはlet構文でのみ多相型を導入できる言語とする。

提案手法の有効性を示すために、Emacs上で簡易言語に対する変数名補完を行うEmacsモードを実装した。

関連研究

変数名補完をEmacs上で行うシステムとして、Java言語のためのEmacsモードであるJDEE [3]がある。JDEEは字下げ、色付け、デバッグ、コンパイル等をEmacs上から行えるようなJavaの統合開発環境であり、フィールド名、メソッド名の補完機能も持つ。フィールド名、メソッド名の補完は変数名をタイプ後にピリオドをタイプし、その後、定められたキーを押すことにより補完が行われるというものである。変数は宣言時にクラス名を記述するため、このクラス情報から容易にこれらの名前を取得することができる。本研究では型推論機構を持つ言語を対象とし、型推論を行

うことにより補完候補となる変数の絞り込みを行う。このような試みは著者らの知る限りこれまでには行われていない。

本論文の構成

本論文の構成は以下のようになっている。まず 2 章において、簡易な静的型付き言語を定め、本提案手法における補完計算の流れを示す。3 章において構文解析手法について述べる。4 章において構文木の補完方式について述べる。5 章において型推論方式を提示する。6 章において補完候補の絞り込みを行う。7 章において Emacs モードによる実装について述べる。8 章においてまとめと今後の課題を述べる。

2 補完について

変数名の補完候補計算は以下の流れで行う。

1. カーソル位置まで構文解析を行う
2. 構文解析の終了時の情報からダミー構文木を補完することにより具象構文木を作成する
3. 具象構文木を抽象構文木に変換する
4. カーソル位置の型およびカーソル位置で有効な変数集合およびそれらの変数の型を推論する
5. 型および入力中の文字列による候補の絞り込みを行う

本論文で用いる言語の構文を以下のように定める。型システムについては通常の ML 系の let 多相の型システムとし、定義は省略する。

$$
\begin{array}{lcll}
\mathit{start} & ::= & \mathit{exp} & (1)\\
\mathit{exp} & ::= & \mathit{appexp} & (2)\\
 & | & \texttt{fn}\ \mathit{id}\ \Rightarrow \mathit{exp} & (3)\\
\mathit{appexp} & ::= & \mathit{atexp} & (4)\\
 & | & \mathit{appexp}\ \mathit{atexp} & (5)\\
\mathit{atexp} & ::= & \mathit{id} & (6)\\
 & | & \mathit{const} & (7)\\
 & | & (\mathit{exp}) & (8)\\
 & | & \texttt{let val}\ \mathit{id} = \mathit{exp}\ \texttt{in}\ \mathit{exp}\ \texttt{end} & (9)
\end{array}
$$

右側に振ってある番号は表 2 における r6, r7 等の還元の番号に対応する。(1) はプログラムの開始を表す規則、(3) は関数、(5) は関数適用、(6) は変数、(7) は定数、(8) は結合順位を変更するための括弧、(9) は変数束縛の構文である。(2)、(4) は関数適用が左結合であることを構文規則に反映させたものである。

変数名補完計算の流れを以下に示す。以下のようなソースコードを入力していたとする。

```
let val x = 1 in
    let val y = fn x => fn y => x y in
        let val z = fn x => x in y _
```

_はカーソル位置を表す。この状況でカーソル位置で有効な変数は x と y の二つであり、カーソル位置は変数 y の後なので、ここに変数が来る場合は、その変数は関数適用の引数の式である。変数以外の可能性としては、開き括弧 (、キーワード end、let、定数がある。補完候補となるのは、カーソル位置で有効な変数であり、x, y, z の 3 つである。これらの変数の型は

$$
\begin{array}{lcl}
\texttt{x} & : & \mathrm{int}\\
\texttt{y} & : & \forall\alpha,\beta.\ (\alpha \to \beta) \to \alpha \to \beta\\
\texttt{z} & : & \forall\alpha.\ \alpha \to \alpha
\end{array}
$$

である。変数 y の型は関数を引数にとる関数型であるため、カーソル位置に変数が入力される場合には、その変数の型は関数型でなければならない。3つの変数のうち関数型を持つのは y と z の2つであり、x は該当しない。よって候補となるのは y と z の2つである。

3 構文解析

変数名補完の候補を求めるために最初に行うことは構文解析である。構文解析を行うことでソースコードを具象構文木へと変換する。構文解析の対象とするのは編集中のソースコードの最初から、現在入力中か、あるいは入力を始めようとしている字句の前までを構文解析の対象とする。以降では、特に明記しない限り現在入力中か入力を始めようとしている字句の前をカーソル位置とする。また、字句解析器はカーソル位置まで来たら EOF を返すこととする。字句は以下のものを用いる。

$$\texttt{let},\ id,\ \texttt{val},\ \texttt{in},\ \texttt{end},\ =,\ \Rightarrow,\ \texttt{fn},\ (,\ ),\ const,\ \text{EOF}$$

id は識別子のための字句であり、識別子の綴りを属性として持つ。$const$ は定数のための字句であり、定数を属性として持つ。属性情報が必要な場合、属性を添字として $id_{\texttt{x}}$、$const_2$ のように明示する。

構文解析器は、EOF を初めて先読みした時、現在の状態において、$atexp$ に還元される字句 id を先読みすることができる状態であった場合、属性としてカーソル位置であるという情報を持つ id という字句を EOF の前に挿入し、それを先読みする。以下ではこの字句を id_{cursor} と記述する[1]。let 式、関数式で束縛する変数部分では変数名補完は行わないので、それを除外するために、$atexp$ に還元される字句 id のみを対象とする。$atexp$ に還元される字句 id が構文定義上来ることが出来ない場合は変数名補完はできないので、この時点で変数名補完の計算を終了する。

2回目の EOF の先読み時には、現在の構文解析器のスタックの状態と、その時点までの還元の履歴を返して終了する。スタックには構文解析時に還元されなかった終端記号（字句）および非終端記号の情報、および状態情報がある。状態情報は以降の計算に不要であるので、状態情報を取り除いた終端記号、非終端記号の列を返す。

構文解析器の動作の変更を行う理由は、EOF で還元を行ってしまうと、挿入した字句 id より後の部分にプログラムがないという前提で還元が行われてしまうため、それを防ぐためである。

簡易言語での状態遷移表は表 1 のようなものになる。

例えば、以下のようなソースコードを入力しているとする。

```
let val x = 2 in x _
```

_はカーソルのある位置とする。

この状態で構文解析を行い、終了時点のスタック情報と、終了時点までに還元をすることによって構築された部分的な構文木情報を結果として得る。この例は補完の候補がない例であるが、構文解析木のサイズを小さくするためこの例で説明する。

スタックから状態番号を取り除いたものは、

$$\texttt{let}, \texttt{val},\ id_{\texttt{x}}, =,\ exp,\ \texttt{in},\ appexp$$

であり、構文木は図 1 のような構文木が得られる。

[1] プログラム中に cursor という識別子が現れる場合はそれに対する字句は $id_{\texttt{cursor}}$ のように添字はタイプライタフォントで記述し区別する。

	id	*const*	`fn`	⇒	`(`	`)`	`let`	`val`	=	`in`	`end`	EOF
0	s15 r6	s16 r7	s6		s1		s7					
1	s15 r6	s16 r7	s6		s1		s7					
2	s15 r6	s16 r7	s6		s1		s7					
3	s15 r6	s16 r7	s6		s1		s7					
4	s15 r6	s16 r7	s6		s1		s7					
5	s15 r6	s16 r7	r2	r2	s1	r2	s7	r2	r2	r2	r2	r2
6	s9											
7								s11				
8												acc
9				s2								
10						s20 r8						
11	s12											
12									s3			
13										s4		
14											s22 r9	
15	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6	r6
16	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7	r7
17	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1	r1
18	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4	r4
19	r5	r5	r5	r5	r5	r5	r5	r5	r5	r5	r5	r5
20	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8	r8
21	r3	r3	r3	r3	r3	r3	r3	r3	r3	r3	r3	r3
22	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9	r9

	start	*exp*	*appexp*	*atexp*
0	g8	g17 r1	g5	g18 r4
1	g8	g10	g5	g18 r4
2	g8	g21 r3	g5	g18 r4
3	g8	g13	g5	g18 r4
4	g8	g14	g5	g18 r4
5	r2	r2	r2	g19 r5
6				
7				
8				
9				
10				
11				
12				
13				
14				
15	r6	r6	r6	r6
16	r7	r7	r7	r7
17	r1	r1	r1	r1
18	r4	r4	r4	r4
19	r5	r5	r5	r5
20	r8	r8	r8	r8
21	r3	r3	r3	r3
22	r9	r9	r9	r9

表 1. 簡易言語の構文解析のための状態遷移表

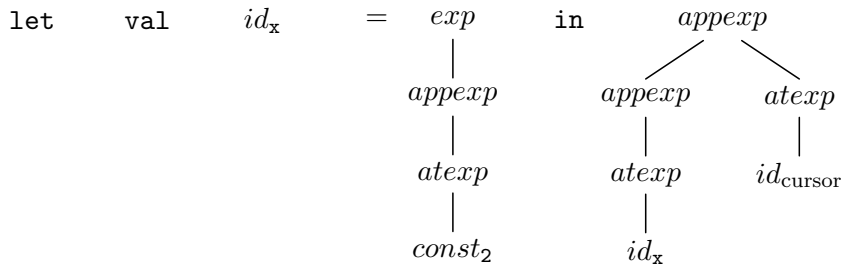

図 1. 構文解析終了時点の構文木

4 構文木の補完

構文解析の結果得られる構文木は終了時点までの還元により構築された部分的な構文木である。部分的な構文木から、ダミー構文木を使って完全な構文木を構築する。構文木の補完には一般に無限の可能性があるが、本論文では有限個生成することとし、その生成方針について本節で述べる。

ダミー構文木は [] で表すこととし、ダミーは任意の非終端記号および属性値を持つ終端記号 (字句) の部分に用いるものとする。ダミーに成り得る記号は、各構文規則の右辺において何らかの非終端記号かあるいは終端記号 id より右側に現れる記号とする。

ダミー構文木を含む構文規則を以下のように定義する。

$$
\begin{array}{lcl}
start & ::= & exp \\
exp & ::= & appexp \\
 & | & \mathtt{fn}\ id\ \Rightarrow exp \\
 & | & [\,] \\
appexp & ::= & atexp \\
 & | & appexp\ atexp \\
atexp & ::= & id \\
 & | & const \\
 & | & (exp) \\
 & | & \mathtt{let\ val}\ id = exp\ \mathtt{in}\ exp\ \mathtt{end} \\
 & | & [\,]
\end{array}
$$

$start$,$appexp$, id は各構文規則の右辺において、何らかの非終端記号かあるいは終端記号 id の右側に現れることがない。

構文木を補完して完全な構文木にするために、構文解析終了時のスタック情報を用いる。構文木の補完手法について、前節の構文解析の具体例で説明する。

補完を行うための手順を以下に示す。この手順においては受け取ったスタックを用いて、ダミー要素を補いながら還元を行う。生成規則は 2 章の言語の定義において、::= を $\rightarrow$ に読み替え、(1) から (9) をそのまま用いる。開始記号は $start$ とする。以下において、非終端記号 1 つを表すメタ変数として A、終端または非終端記号の列 (空列 ϵ を含む) を表すメタ変数として α、終端または非終端記号 1 つを表すメタ変数として s を用いる。スタックの記号列は、スタックトップから順に、$s_1, \ldots, s_n$ であるとする。

(1) s_1 が開始記号 $start$ ならば終了。s_1 が開始記号 $start$ でなければ、s_1 を右辺に含む各生成規則 $A \rightarrow \alpha$ において、右辺 α を s_1 で分離し、$\alpha = \alpha_1 s_1 \alpha_2$ とする。α_1 がスタックトップからの記号列と一致する、すなわち $\alpha_1 = s_i \ldots s_2$ となるものについて還元を行う。還元時には、α_2 の中の非終端記号をダミー要素に変えた記号列をスタックに積んでから還元を行う。還元後のスタックの記号列は、スタックトップから順に $A s_{i+1} \ldots s_n$ である。

(2) (1) に戻る。

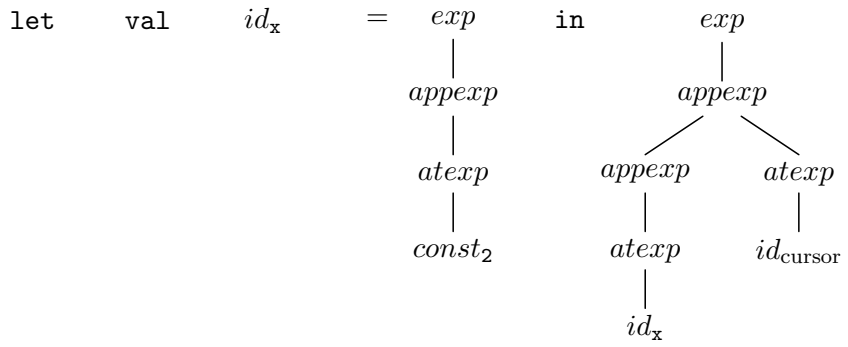

図 2. 図 1 の構文木に規則 2 を適用して得られる構文木

この手法では一般に還元が無限に起こる還元系列が存在するが、本論文ではしきい値を決め、各還元系列において深さがある一定値になったら打ち切ることとする。これにより、構文木の補完は有限時間で終了する。

3 章で挙げた具体例の構文解析の結果の構文木とスタック情報を入力としてこの手順で補完を行うと以下のようになる。今、スタックトップは *appexp* である。この非終端記号 *appexp* が現れる還元規則は規則 2 と規則 5 であり、また規則 2,5 において *appexp* より左には記号が存在しないため両方の規則で還元が可能である。規則 2 を構文木 (図 1) に適用すると図 2 の構文木が得られ、規則 5 を適用すると図 3 の構文木が得られる。

規則 5 では規則中の *atexp* がスタックトップの記号の右側にあるため、*atexp* の部分はダミー要素とする。この時点におけるスタックの状態は、規則 2 を適用した結果得られる構文木 (図 2) に対応するスタックは

$$\mathtt{let},\ \mathtt{val},\ id_{\mathtt{x}},\ =,\ exp,\ \mathtt{in},\ exp$$

であり、規則 5 を適用した結果得られる構文木 (図 3) に対応するスタックは

$$\mathtt{let},\ \mathtt{val},\ id_{\mathtt{x}},\ =,\ exp,\ \mathtt{in},\ appexp$$

である。これら 2 つのうち、図 3 に対応するスタックのトップは再度 *appexp* となっている。この後、規則 5 が再度適用できるので、規則 5 を何度でも適用できることになる。図 2 に対応するスタックのトップは *exp* である。*exp* が含まれる還元規則を列挙すると規則 1、3、8、9 であるが、規則 9 だけが適用可能である。規則 9 では *exp* が 2 回現れているが、左側の *exp* までの記号列はスタックトップからの記号列と一致せず、右側の *exp* までの記号列に一致する。`end` を補うことにより構文木 (図 2) に規則 9 を用いて還元を行うと最上位のノードは let の規則により *atexp* へと還元され、スタックは以下のようになる。

$$atexp$$

atexp を含む規則は規則 4 と規則 5 であるが、規則 5 はスタック上に *appexp* がないため一致しないので規則 4 を用いて還元することとなる。規則 4 を用いて還元を行うと最上位ノードは *appexp* となり、スタックは、

$$appexp$$

となる。ここで還元される規則は上で *appexp* が出てきた時と同じように規則 2 と規則 5 である。規則 5 については今回も省略する。規則 2 で還元を行うと最上位ノードは *exp* になり、スタックは次のようになる。

$$exp$$

exp を含み、このスタックと一致するのは規則 1 だけであるので、それを適用すると図 4 のようになる。スタックトップが開始記号 *start* となり、一つの構文木が完成したことになる。以上の手

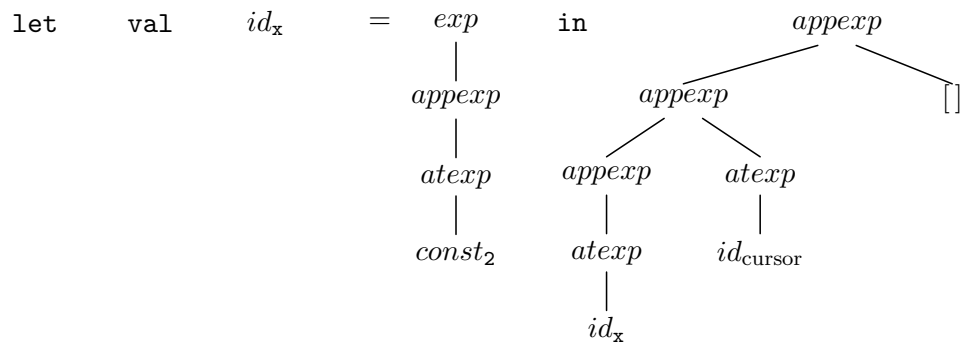

図 3. 図 1 の構文木に規則 5 を適用して得られる構文木

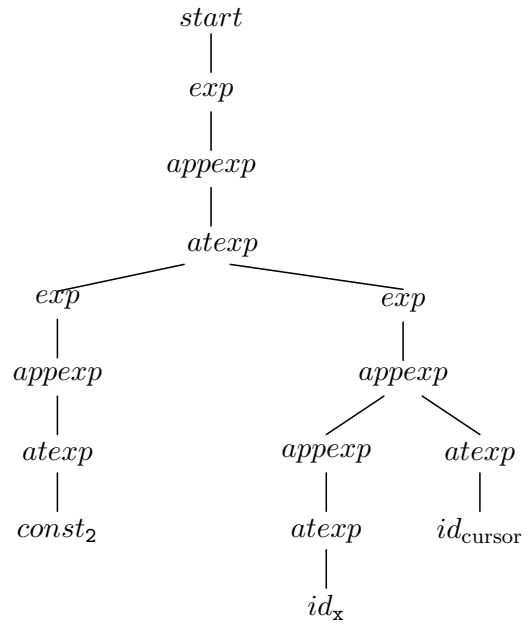

図 4. 開始記号まで還元して得られる構文木

続きでは *appexp* による循環があるが、深さをある一定値で打ち切るので、構文木の補完処理は有限時間で終了する。

本論文の簡易言語では構文木の補完で枝分かれが発生するのは規則 2 と 5 の部分のみである。規則 5 の適用が必要な例を以下に挙げる。

```
let val f = fn x => + x 1 in
    f (f_
```

_はカーソル位置を表す。+はこの例の説明のために定数として追加したものであり、int → int → int 型の定数である。この状態で右側に) と end を補った場合は型エラーになる。カーソル位置の右側にダミー要素 [](*atexp* に対応) を 1 つ補ったのちに) と end を補うと型検査を通過する。

5 型推論

ここまでの段階で、一般に複数の抽象構文木が作られるが、その全てに対し型推論を行う。

型推論を行う前に、4 章で補完により構築した具象構文木を以下の抽象構文木に変換する。

$$
\begin{array}{rcl}
e & ::= & x \\
& | & c \\
& | & \textbf{cursor} \\
& | & e\ e \\
& | & \lambda x.\ e \\
& | & \textbf{let}\ x\ =\ e\ \textbf{in}\ e \\
& | & [\,]
\end{array}
$$

x は変数、c は定数、e は式を表すためのメタ変数である。なお、本論文の簡易言語では、λ, **let** による束縛部分でダミーの変数が必要になる場合はない。

具象構文木から抽象構文木への変換は自明であり省略する。 型推論のアルゴリズムは Milner のアルゴリズム $\mathcal{W}$ [7] をもとにダミー要素、カーソル要素を追加し、型を破壊的に書き換えるように変更したものを用いる。また、結果としてはカーソル要素の型およびカーソル位置で有効な変数からそれらの型への対応関係を返す。このアルゴリズムを図 5 に示す。ダミー要素、カーソル要素は新しい型変数として型推論を行う。$\mathcal{U}$ は単一化関数であり、結果として、変数から、参照を使った型への対応関係を返す。Cls は型環境 Γ と型 τ を受け取り、τ の中に現れる型変数のうち Γ の中に自由に現れる型変数を覗いた型変数を束縛して得られる型への参照を返す関数である。

型は以下のように定義する。

$$
\begin{array}{rcl}
\sigma & := & \forall\alpha_1 \ldots \alpha_n.\tau \\
\tau & := & \mathrm{int} \\
& | & \alpha \\
& | & \tau \to \tau
\end{array}
$$

σ が多相型、τ が単相型である。本論文では、int 型、型変数、関数型からなる型のみを扱う。

本論文の型推論アルゴリズムでは、効率面を考慮して型を破壊的に書き換えることとし、型の定義として以下のように参照を使ったものを用いる。

$$
\begin{array}{rcl}
\tau & := & \uparrow \mathrm{int} \\
& | & \uparrow \alpha \\
& | & \uparrow (\tau \to \tau)
\end{array}
$$

置換の適用は以下のように定義し、破壊的に型を変更する。

$$
\begin{array}{rcll}
subst\ (S, \tau) & = & \tau\uparrow := \mathrm{int} & (\tau\uparrow \text{が int のとき}) \\
& & \tau\uparrow := S(\tau) & (\tau\uparrow \text{が}\alpha\text{のとき}) \\
& & subst(S, \tau_1);\ subst(S, \tau_2) & (\tau\uparrow \text{が}\tau_1 \to \tau_2\text{のとき})
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{C}(\Gamma, e) \ = & \mathit{cursorType} := \text{nil};\ \mathit{cursorEnv} := \Gamma; \\
& \mathcal{C}'(\Gamma, e); \\
& \text{return } (\mathit{cursorType},\ \mathit{cursorEnv}) \\
\mathcal{C}'(\Gamma, \mathit{cursor}) = & \mathit{cursorType} :=\uparrow \beta;\ (\beta \text{ fresh}) \\
& \mathit{cursorEnv} := \Gamma;\ \ \text{return } \mathit{cursorType} \\
\mathcal{C}'(\Gamma, x) = & \Gamma(x) = \forall\alpha_1 \ldots \alpha_n.\tau_1 \text{ のとき、} \\
& \qquad \text{return } [\beta_1/\alpha_1] \ldots [\beta_n/\alpha_n]\tau_1\ (\beta_1, \ldots, \beta_n \text{ fresh}) \\
& \Gamma(x) = \tau_1 \text{ のとき、 return } \tau_1 \\
\mathcal{C}'(\Gamma, c) = & \text{return} \uparrow \text{int} \\
\mathcal{C}'(\Gamma, e_1\ e_2) = & \tau_1 = \mathcal{C}'(\Gamma, e_1);\ \tau_2 = \mathcal{C}'(\Gamma, e_2); \\
& S = \mathcal{U}(\tau_1, \uparrow (\tau_2 \rightarrow \uparrow \beta));\ (\beta \text{ fresh}) \\
& \mathit{subst}(S, \tau_1);\ \mathit{subst}(S, \tau_2); \\
& \text{return } S(\beta) \\
\mathcal{C}'(\Gamma, \lambda x.e) = & \tau_1 = \mathcal{C}'(\Gamma\{x :\uparrow \beta\}, e);\ (\beta \text{ fresh}) \\
& \text{return} \uparrow (\uparrow \beta \rightarrow \tau_1) \\
\mathcal{C}'(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) = & \tau_1 = \mathcal{C}'(\Gamma,\ e_1); \\
& \sigma = \mathit{Cls}(\Gamma, \tau_1); \\
& \text{return } \mathcal{C}'(\Gamma\{x : \sigma\},\ e_2) \\
\mathcal{C}'(\Gamma, [\,]) = & \text{return} \uparrow \beta\ (\beta \text{ fresh})
\end{array}
$$

図 5. ダミー要素を含む抽象構文に対する型推論アルゴリズム $\mathcal{C}$

6 補完候補の絞り込み

5 章で提示した型推論アルゴリズム $\mathcal{C}$ により、カーソル位置で有効な変数からその型への対応関係およびカーソル位置の型が得られる。カーソル位置で有効な変数それぞれについて、変数に対応付けられている型とカーソル位置の型の単一化を行い、単一化が成功した変数を残し、失敗した変数を除外する。カーソル位置の型が τ_1 であり、注目している変数に対応付られている型が $\forall\beta_1 \ldots \beta_n.\ \tau_2$ であるとき、τ_1 と $\tau_2' = [\alpha_1/\beta_1] \cdots [\alpha_n/\beta_n]\tau_2\ (\alpha_1, \ldots, \alpha_n \text{ fresh})$ で単一化 $\mathcal{U}\ (\tau_1, \tau_2')$ を行う。変数に対して、現在入力中の字句が先頭の部分文字列になっている変数を候補とする。

例えば、以下のようなプログラムを入力しているとする。

```
let val ya = fn x => x in
  let val xb = fn x => x in
    let val xc = 1  in
      (fn x => x 1) x_
```

_はカーソル位置を表す。カーソルより後に end を 3 つ補って出来た構文木を元に補完候補の計算を行う場合を考える。このとき、カーソル位置の型およびカーソル位置で有効な 3 つの変数 ya, xb, xc の型はそれぞれ以下のように推論される。

$$
\begin{array}{l}
\textbf{cursor} : \text{int} \rightarrow \alpha \\
\texttt{ya} : \forall\gamma.\ \gamma \rightarrow \gamma \\
\texttt{xb} : \forall\gamma.\ \gamma \rightarrow \gamma \\
\texttt{xc} : \text{int}
\end{array}
$$

これらの 3 つの変数に対しそれぞれカーソル位置の型と単一化を行うと ya と xb は単一化が成功し、xc は単一化が失敗するので、変数 xc は除外され、残るのは ya と xb の 2 つである。現在 x までを入力しているので、最終的に候補として提示されるのは xb となる。

7　実装

これまでに提示した手法に基づき、Emacs[2] 上で変数名補完を行うための Emacs モード lambda-mode を作成した。作成した lambda-mode は web ページ (`http://www.cs.ise.shibaura-it.ac.jp/complement/`) から取得できる。この章では実装について述べたのち、変数名補完にかかる時間を 6 章の具体例で計測した時間を述べる。

lambda-mode では変数名補完機能だけでなく簡易な字下げ機能も実装した。

変数名補完の処理は

1. プログラマが打鍵した際に候補を計算
2. 候補をプログラマに提示
3. プログラマが選んだ候補で補完

の 3 つに分けられる。2 と 3 に当たる処理は既存の auto-complete.el [1] という入力補完を行うための Emacs モードを利用する。

auto-complete.el は補完の候補となるリストを元にプログラマが 1 文字打つ度に予め設定された候補を求める関数を呼び出し、候補があれば図 6 のように画面に表示する。そこから Emacs 利用者は候補を選択して補完を行うか、あるいは表示された候補を無視して入力を続けることもできる。

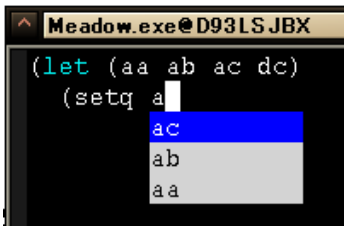

図 6. auto-complete による補完の例

auto-complete モードはデフォルトでは編集中のテキストを単語単位で区切ったものを候補とするが、候補を作成する関数を書き換えることができる。補完候補となる変数名のリストを求める関数で候補作成関数を書き換える。変数名補完は lambda-mode をメジャーモードとし、auto-complete モードをマイナーモードとして実現している。

字句の定義は、*id* は大小の英字の列、*const* は 0 以外の数字から始まる 0 から 9 までの数字列とした。さらに、int 型の四則演算+, -, /, *を定数に追加した。

構文解析器は手書きで Emacs Lisp で実現し、構文解析器の用いる状態遷移表は yacc と互換性のある kmyacc[4] を用いて作成した表をもとに EOF を受け取った際の動作を書き換えたものを用いた。3 章で述べたように構文解析では入力している字句の直前までを構文解析の対象とするため、構文解析開始時にカーソル位置を現在入力している字句の直前に移動し、構文解析終了後、カーソルを元の位置に戻す。ただし、現在字句を一文字も入力していない状態ではカーソルの位置は移動しない。

4 章で提示した補完手順では還元規則においてスタックトップにある記号が出現した位置から左側にある記号列を全て比較していたが、今回用いている簡易言語ではスタックトップと 2 番目までを見ればどの還元規則を適用すれば良いかが決定できるので実装では 2 番目までを比較の対象とした。スタックトップと 2 番目の記号からどの規則を適用すれば良いかを調べることが出来る表を作成した。表の作成方法を以下に示す。

(1) スタックの 1 番目が *start* であるますには acc と書く。

(2) スタックの 1 番目にある記号 s_1 が *start* でない場合、s_1 が右辺に現れる構文規則を列挙する。

(3) 列挙した構文規則の中の記号 s_1 のひとつ左にある記号を s_2 とする。s_1 が構文規則中で一番左にある場合は s_2 は空となる。

(4) (2) で列挙した構文規則について s_2 が空でない規則それぞれについて以下を実行する。

(4-a) 表中のスタックの 1 番目が s_1、2 番目が s_2 のますに (2) で列挙した構文規則の番号を書く。このとき構文規則が複数ある場合は複数書き込む。

(5) (2) で列挙した構文規則について s_2 が空である規則それぞれについて以下を実行する。

(5-a) スタックの 1 番目が s_1 である行の空いているます全てにその構文規則の番号を書く。

以上の手順で本論文の簡易言語に対する表を作成する場合について考える。例えばスタックの 1 番目が非終端記号 *appexp* である場合は、$s_1 = appexp$ となる。*appexp* は生成規則 2 と 5 に現れる。規則 2 の中で *appexp* の左には記号がないので s_2 は空となり、表の *appexp* の行の全てのますに 2 を書く。規則 5 の中でも *appexp* の左には記号がないので同様に行の全てのますに 5 を書く。同様に全ての記号について上記手順を行うことにより、どの還元規則を適用するかをスタックトップと 2 番目の要素のみから判別するための表 2 が得られる。この表を用いて構文木の補完を行う手順を以下に示す。

(1) スタックの 1 番目を s_1、2 番目を s_2 とする。

(2) 表で 1 番目が s_1、2 番目が s_2 であるますに acc が書かれていたら構文木が完成しているので終了する。

(3) 表で 1 番目が s_1、2 番目が s_2 であるますに書かれている番号に対応する構文規則について、s_1 を含め s_1 の左側にある記号の数を求める。ますに書かれている構文規則の個数が複数の場合、それぞれについて以下の処理を実行する。

(3-a) 求めた記号の数だけスタックから記号をポップし、スタックの最上位に適用した構文規則の左側にある非終端記号をプッシュする。

(4) (1) へ戻る。

構文木の補完に関しては以上である。

変数名補完の候補の計算が始まるタイミングであるが、auto-complete.el のデフォルトの動作である文字を入力したときに計算が始まり候補が表示される仕組みとなっている。

Emacs lisp では関数呼び出しの深さの上限 max-lisp-eval-depth および変数の束縛数の上限 max-specpdl-size が設定されており、それぞれデフォルトでは 400, 1000 である。構文木補完の深さの閾値によってはそれら上限を超えてしまうことがあるため、lambda-mode では補完候補計算の間のみそれらの設定値を大きめに変更する。

作成した lambda-mode の動作例を図 7 に示す。

作成した lambda-mode に補完候補の計算にかかった時間を計測した。計測するのは補完候補計算開始時点から終了時点までとした。この時間はほぼ、Emacs 利用者がキーを打鍵してから補完候補が表示されるまでの時間である。計測時には構文木の補完を行う際の深さの閾値を 15 とした。6 章のプログラム例で候補計算にかかる時間は 63 ミリ秒であった。測定した際の環境は、OS は Windows XP Professional SP3、CPU は Intel Core 2 Duo E7300 2.6GHz、メモリは 3.25GByte であり、Emacs は Meadow(GNU Emacs 22.2.1) を使用した。

このデータからは本研究の提案手法の実用性を示したとまでは言えないが、本論文の提案手法は素朴なものであり、少なくとも非常に小規模のプログラムについては有効に用いることができるということは示された。

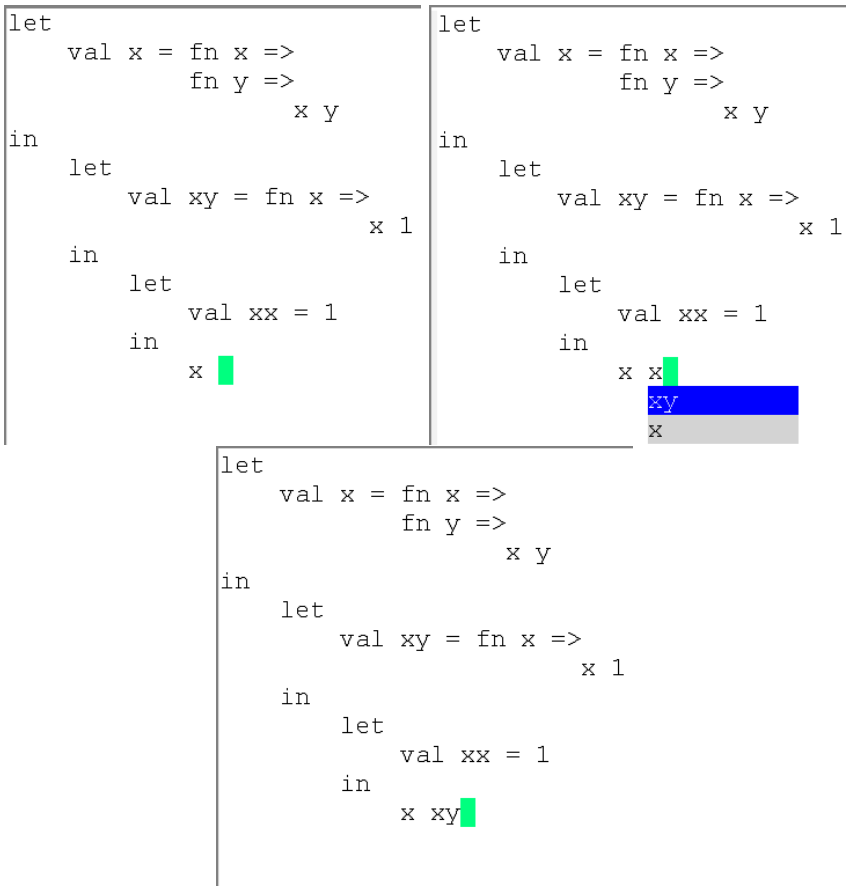

図 7. lambda-mode による補完の例

スタック		2 番目											
		空	*id*	*const*	`fn`	⇒	`(`	`)`	`let`	`val`	`=`	`in`	`end`
1 番目	*start*	acc	acc	acc	acc	acc	acc	acc	acc	acc	acc	acc	acc
	exp	1	1	1	1	3	8	1	1	1	9	9	1
	appexp	5,2	5,2	5,2	5,2	5,2	5,2	5,2	5,2	5,2	5,2	5,2	5,2
	atexp												
	`fn`	3	3	3	3	3	3	3	3	3	3	3	3
	id	6	6	6	3	6	6	6	6	9	6	6	6
	const	7	7	7	7	7	7	7	7	7	7	7	7
	⇒				3								
	`(`	8	8	8	8	8	8	8	8	8	8	8	8
	`)`												
	`let`	9	9	9	9	9	9	9	9	9	9	9	9
	`val`								9				
	`=`		9										
	`in`												
	`end`												

スタック		2 番目		
		exp	*appexp*	*atexp*
1 番目	*start*	acc	acc	acc
	exp	1	1	1
	appexp	5,2	5,2	5,2
	atexp		5	
	`fn`	3	3	3
	id	6	6	6
	const	7	7	7
	⇒			
	`(`	8	8	8
	`)`	8		
	`let`	9	9	9
	`val`			
	`=`			
	`in`	9		
	`end`	9		

表 2. 還元規則適用表

8　まとめと今後の課題

本研究では本研究は型情報を考慮した変数名補完の実用化へ向けた第一歩であり、簡易言語上で補完方式を提示した。簡易言語は、後方の変数参照のない、let 多相の言語とした。本研究の方式は素朴に構文木を可能なすべての方法で補ってから型推論を行うというものであり、無限個の構文木の生成を回避するために構文木生成の深さの閾値を設定している。

変数名補完方式は以下のような点で改良の余地が残されている。

- 現在は１文字打つたびに一から計算を行っているが、適切に計算結果を保存することにより、型推論および構文解析について計算結果を部分的に再利用することが可能であると考えられる。型推論をインクリメンタルに行う手法が Aditya らによって提案されている [6]。プログラムの一部を変更した場合に変更した部分とその影響のある部分のみ型推論を行う手法であり、型推論計算の再利用を行うものである。この研究はトップレベルの宣言の単位で型推論計算を行えるようにしたものであり、本研究において、この研究を型推論の計算の再利用に有効に用いることができると期待される。
- 現在、構文木の補完時には、深さの閾値を設定することにより無限個の構文木の生成を回避しているが、これではすべての場合が尽くされない。型推論フェーズとうまく連携することにより、必要十分な構文木補完を有限時間で行えるようになることが期待される。
- Haskell 等の言語においては、変数の束縛より前にその変数を使用できる。また、ML 系言語においても相互再帰関数においては束縛より前に変数の使用ができる。これらの対処法を考察する。
- 現在は型の annotation を許していないが、これを許すように拡張する。
- 現在は変数のみを補完対象としているが、キーワードの補完も行えるように拡張する。

将来これらの改良を行うことにより、実際の実用言語において実用的に使える補完方式の実現を目指している。

謝辞

本論文について大変有益な意見を下さった査読者の方々に感謝します。本研究の一部は科学研究費補助金の補助を得て行われた。

参考文献

[1] EmacsWiki: Auto complete. `http://www.emacswiki.org/emacs/AutoComplete`.

[2] GNU Emacs. `http://www.gnu.org/software/emacs/`.

[3] Java development environment for Emacs. `http://jdee.sourceforge.net/`.

[4] kmyacc. `http://www005.upp.so-net.ne.jp/kmori/kmyacc/`.

[5] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 379–405, Cambridge, USA, August 1991.

[6] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.