LR 構文解析のエラー回復機能を用いたキーワード補完機能の系統的導出

白 楊[1,a)]　篠埜 功[1]

概要：キーワード補完とは様々な言語の開発環境において提供されている機能であり、入力中の文字列を接頭辞に持つキーワードをポップアップウィンドウ等に表示するものである。キーワード補完機能を用いることによりキーワード入力時間や綴りミスを削減することができる。本発表では、様々な言語に対して仕様が明示されたキーワード補完機能を提供することを目的とする。入力中の構文が不完全なソースコードに対応するため、Yacc による LR 構文解析の誤り回復機能を用いることとし、キーワード補完機能の実装を仕様から系統的に導出する手法を提案する。カーソル位置の（入力途中の）キーワードの字句を識別するため、入力途中であることを示す字句を字句解析および構文解析の仕様記述に追加する。本提案手法においては構文解析を行うため、補完候補計算において構文に関する文脈が考慮される。実装の導出は Byacc という構文解析器生成系のソースコードを用いて行う。Yacc の構文規則記述およびユーザが記述したキーワードを入力とし、構文規則の記述への字句 error の機械的挿入により補完候補計算プログラムの自動生成を行うツールを作成した。例として、C 言語の Yacc の仕様記述ファイルを入力とし、C 言語のキーワードの補完を行うプログラムを生成した。現状では字句の仕様記述にはカーソル位置を表す字句を手動で追加する必要がある。補完対象キーワードは自動生成ツールの使用者が指定できる。

キーワード：keyword completion, error recovery, automatic generation

Derivation of keyword completion programs using the error recovery in LR parsing

Yang Bai[1,a)]　Isao Sasano[1]

Abstract: The keyword completion, which is a functionality for popping up keywords starting with a string currently being input, is provided in the IDEs for various languages. The keyword completion reduces the time for inputting keywords and spelling errors. In this presentation we aim at providing a keyword completion whose specification is given explicitly. We use the error recovery in LR parsing in order to cope with incomplete program text and systematically derive an implementation of keyword completion from a specification. In order to identify the incomplete keywords in the cursor position, we add a token, indicating it is currently input, to the specification of the lexer and the parser. We parse the program being input so that the syntactic context is taken into account in computing candidates. We systematically derive an implementation of the keyword completion by using a parser generator BYacc. We have implemented a tool that generates programs computing candidates by inserting a special token error to the Yacc specification. The tool takes as its input keywords listed by the users and a Yacc specification. As an example, we generate a program for completing keywords in programs in the C language from a Yacc specification. Currently we need to manually add a token for the cursor position in the specification of the lexer. Keywords to be popped up are specified by the tool user.

[1] 芝浦工業大学大学院　理工学研究科
Graduate School of Engineering and Science, shibaura Institute of Technology
[a)] ma14501@shibaura-it.ac.jp

1. はじめに

キーワード補完とは様々な言語の開発環境において提供

されている機能であり、入力中の文字列を接頭辞に持つキーワードをポップアップウィンドウ等に表示するものである。キーワード補完機能を用いることによりキーワード入力時間や綴りミスを削減することができる。広く使われている Eclipse や Visual Studio 等における様々な言語に対する開発環境において識別子補完、キーワード補完など様々な入力補助機能が提供されているが、入力補完においては作成中の不完全なソースコードに対して補完候補を柔軟に計算することが求められ、現状では仕様を定めないか、あるいは公開せずに補完機能が提供されている。

本研究では、様々な言語に対して仕様が明示されたキーワード補完機能を提供することを目的とする。入力中の構文が不完全なソースコードに対応するため、Yacc による LR 構文解析の誤り回復機能を用いることとし、キーワード補完機能を仕様から系統的に実装を導出する手法を提案する。Yacc の誤り回復は、構文規則の記述における **error** という特別な字句の用い方により制御することができる。また、カーソル位置の（入力途中の）キーワードの字句を識別するため、（接頭辞を共有する）キーワード毎に入力途中であることを示す字句を字句解析および構文解析の仕様記述に追加する。本提案手法においては構文解析を行うため、補完候補計算において構文に関する文脈が考慮される。

実装の導出は Byacc[1] という構文解析器生成系のソースコードを用いて行う。Yacc の構文規則記述およびユーザが記述したキーワードを入力とし、各キーワードに対して入力中であることを示す字句の追加および構文規則の記述への字句 **error** の機械的挿入により自動生成を行う。

本論文の構成は以下の通りである。2 節において提案手法におけるキーワード補完計算の流れを示す。3 節において Yacc の誤り回復機能を用いた構文木の生成について述べる。4 節において、得られた構文木からの補完候補の計算について述べる。5 節において、補完候補計算プログラムの自動生成アルゴリズムを提示する。6 節において、自動生成プログラムに与えるユーザが記述する入力について述べる。7 節において自動生成アルゴリズムに基づいた実装について述べる。8 節において考察、9 節において関連研究について述べる。10 節においてまとめと将来の課題について述べる。

2. 提案手法におけるキーワード補完計算の流れ

ここでは C 言語を例として、提案手法におけるキーワード補完計算の流れを示す。補完計算は字句解析、誤り回復機能を伴う構文解析による構文木生成、構文木からの補完候補の計算からなる。

2.1 字句解析

入力中のソースコードおよびカーソル位置の情報をもとに字句解析を行う。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。

```
int main (void){
  i␣ ( ii = 1 ii;
  else ii;}
```

これはキーワード `if` を入力しようとしているときのプログラムである。ここで␣はカーソル位置を表す。このプログラムは以下のような字句列に分解される。

```
int id_main (void){
  ␣ LP id_ii = id_1 id_ii;
  else id_ii;} EOF
```

本研究では識別子の補完は行わないので、カーソル␣が文字 `i` の直後にある場合、キーワード `if` の入力途中であると考える。キーワード `if` を入力中であることを表すため、字句の定義に ␣ という字句を追加する。また、入力中のキーワード以外の部分も一般にプログラムは書きかけの状態であり、上記プログラムにおいては、`1` の直後の閉じ括弧はまだ入力されいない。

2.2 誤り回復機能を用いた構文解析

プログラミング中のソースコードは、入力途中であることにより、通常構文誤りがある。また、打ち間違いや入力のし忘れによる構文誤りがある場合もある。構文誤りに対応するため、構文解析器の生成に広く用いられている Yacc の誤り回復機能を用いることを考える。Yacc においては、構文定義において **error** という特別な字句を用いることができる。字句 **error** の用い方により、構文誤りからの回復の仕方を制御することができる。例えば、以下のような `if` 式に対する構文規則およびアクション記述を考える。

selection_statement
: `if` LP *expression* RP *statement* `else` *statement*
{$$ = *selection_statement*$_{\texttt{if}}$($3, $5, $7); }
| ...

この規則の記号 `RP` の部分に構文誤りがある場合に対応するため、`RP` を字句 **error** で置き換え、`if` をキーワード `if` を入力中であることを表す字句 ␣ で置き換えた以下のような規則を考える。

selection_statement
: ␣ LP *expression* **error** *statement* `else` *statement*

コンパイラにおいては構文誤りがある場合には構文木を生成する必要はないが、構文誤りがある場合にキーワード補完を行うため、上記のような字句 **error** を含む規則のアクション記述部分にも構文木生成のコードを記述する。アクション記述はもとのアクション記述とほぼ同様でよく、例えば上記の構文規則の場合は以下のように書けばよい。

selection_statement
␣ LP *expression* **error** *statement* else *statement*
{$$ = *selection_statement*$_{\texttt{if}}$($3, $5, $7)}
| ...

2.3 構文木からの補完候補の計算

上記のように誤り回復機能を用いて生成された構文木を用いることにより、構文に関する文脈を考慮して補完候補を計算することができる。構文木の中にキーワード入力中であることを示す字句が含まれている場合、その字句で置き換える前の字句を補完候補とする。それが複数ある場合は、すべての字句を補完候補とする。

3. 誤り回復機能

本研究では Yacc[2] の誤り回復機能を利用することにより、書きかけなどによる構文誤りを許し、かつ構文に関する文脈を考慮したキーワード補完計算を行う。この節では Yacc の誤り回復機能について概観したのち、提案手法におけるキーワード補完候補計算への誤り回復機能の適用について具体例を用いて述べる。

3.1 Yacc の誤り回復

LR 構文解析中に状態遷移表にない字句が先読みの字句になった場合、Yacc は誤り処理状態に入る。Yacc では構文定義中で **error** という特別な字句を用いることができる。Yacc に与える構文規則へ字句 **error** をどのように入れるかによって誤りからの回復をある程度制御することができる。また、アクション中で文 `yyerrok;` を記述することにより、構文解析器を通常のモードに強制的に戻すことができる。ただし、本論文では `yyerrork;` は使用しない。

Yacc での誤り回復は以下のように行われる。

(1) 状態遷移表に先読みの字句に対する動作がない場合、誤り処理状態に入る。
(2) 字句 **error** が読める状態になるまで構文解析スタックから状態等をポップする。
(3) 字句 **error** を読み、状態遷移を行う。
(4) 誤り処理状態においては、先読みの字句に対する動作が状態遷移表にない場合、その字句を読み捨てる。字句を 3 つ読んだら（シフトしたら）正常状態へ復帰する。

3.2 Yacc の誤り回復を用いた構文木生成例

2.1 節で提示したプログラム例を構文解析すると、字句 id_1 までは状態遷移表通りに動作する。字句 id_1 まで読んだ後の先読み字句は id_{ii} であるが、状態遷移表にないため、字句 id_1 の直後に字句 **error** が先読み字句として挿入された扱いとしてから誤り処理状態に入る。

字句 **error** を読むことができる状態が構文解析スタックの中に存在していた場合、構文解析スタックをポップして最初にその状態になったときに字句 **error** をシフトし、3 つの字句を読んで（シフトして）構文解析器が正常状態に復帰し、字句 `EOF` を読んだ後構文解析は正常に終了する。

Yacc の構文規則の開始記号が *start* である場合、Yacc によって以下のような EOF を含む規則が追加される。

```
$accept : start $end
```

ここで`$end` は EOF を表す。

4. 補完候補の計算

前節で提示した構文解析により、構文木が生成される。生成される構文木により、カーソル位置の直前の文字列が何らかのキーワードの接頭辞の場合、そのキーワードが補完候補になる。2 節の例

```
int main (void){
  i␣(ii = 1 ii;
  else ii;}
```

では、カーソル位置の直前がキーワード `if` の接頭辞 `i` であり、`if` が補完候補となる。変数名は補完対象外であるので、`ii` は補完対象とはならない。

カーソル位置の直前に入力がない場合、カーソル位置に入り得るすべてのキーワードを補完候補とする。例えば以下のような入力中のプログラムを考える。

```
int main (␣){
  if (ii = 1) ii;
  else ii;}
```

このプログラムではキーワードの接頭辞が入力されていないが、この位置に入り得るすべてのキーワードが補完候補となる。例えば、ユーザが与えるキーワードファイルに `IF ELSE WHILE VOID CHAR FLOAT LP RP` と記述されている場合、上記の入力に対し、補完候補になるキーワードは `void, char, float, (, )` である。上記のプログラム例においてカーソル位置に `if`、`else`、`while` が入った場合には、構文誤りとなるので、キーワード `if`、`else`、`while` は補完候補にはならない。

5. 自動生成アルゴリズム

ここでは、前節までで提示した流れでキーワード補完計算を行うプログラムの一部を自動生成するアルゴリズムを提示する。ユーザが与えるのは、補完対象にするキーワードを列挙したものと、構文定義である。構文定義はユーザが記述するか、あるいは補完対象言語のコンパイラ実装者が記述したものかもしれない。

以下では、構文規則の右辺に非終端記号がある場合とない場合の 2 つに分けて自動生成アルゴリズムを提示する。

5.1 右辺に非終端記号がある場合

構文定義は $m \geq 1$ 個の構文規則からなっており、その i 番目 $(1 \leq i \leq m)$ の構文規則が以下の形で与えられているとする。

$$N_i : s_1 \ \ldots \ s_{n_i}$$

ここで N_i は i 番目の規則の左辺の非終端記号、s_j $(1 \leq j \leq n_i)$ は i 番目の規則の右辺の j 番目の終端記号あるいは非終端記号である。

自動生成アルゴリズムは、i 番目の構文規則に対し、右辺の各記号 s_j がユーザが列挙したキーワードのいずれかと一致する場合、そのキーワードの記号を、キーワードを入力中であることを表す字句 ␣ と置き換えることにより、以下のような新しい構文規則を生成し、構文定義に追加する。

$$N_i : s_1 \ \ldots \ s_{j-1} \ \textvisiblespace \ s_{j+1} \ \ldots \ s_{n_i}$$

この新しく追加した規則をもとに、各 $l(j < l \leq n_i)$ について、記号 s_l が終端記号の場合、それを字句 **error** と置き換えた以下のような構文規則を生成し、構文定義に追加する。

$$N_i : s_1 \ \ldots \ s_{j-1} \ \textvisiblespace \ s_{j+1} \ \ldots \ s_{l-1} \ \textbf{error} \ s_{l+1} \ldots \ s_{n_i}$$

5.2 右辺に非終端記号がない場合

構文定義は $m \geq 1$ 個終端記号で構成する構文規則からなっており、以下の形で与えられているとする。

$$\begin{array}{rcl} N & : & s_{\{1,1\}} \ \cdots \ s_{\{1,n_1\}} \\ & & \cdots \\ & | & s_{\{m,1\}} \ \cdots \ s_{\{m,n_1\}} \\ & & \cdots \end{array}$$

上記 m 個の各構文規則の右辺の記号はすべて終端記号とする。また、各構文規則の右辺にはキーワードが 0 個以上あり、m 個の構文規則全体の右辺にはキーワードは合計 1 個以上あるとする。非終端記号 N を左辺に持つ構文規則の集まりに対し、字句 ␣ のみを右辺に持つ構文規則を以下のように追加する。

$$\begin{array}{rcl} N & : & s_{\{1,1\}} \ \cdots \ s_{\{1,n_1\}} \\ & & \cdots \\ & | & s_{\{m,1\}} \ \cdots \ s_{\{m,n_1\}} \\ & | & \textvisiblespace \end{array}$$

5.3 補完候補を計算する関数の自動生成アルゴリズム

カーソル位置の直前にキーワードの接頭辞を入力してある場合、本ツールはキーワードの接頭辞を読み、カーソル位置で提示するべきキーワードを生成される構文木から補完候補を計算する関数により返す。以下の形で与えられるとする。

$$N_i : s_1 \ \ldots \ k \ \ldots \ s_{n_i}$$

k はユーザが与えたキーワードファイル内のキーワードとする。$s_j (1 \leq j \leq n_i)$ は s_1 と s_{n_1} 間の非終端記号とする。自動生成アルゴリズムはキーワード k をカーソル位置を意味する ␣ と置き換えた構文規則を追加する。追加される構文規則に対するアクションは以下のように生成する。

$$\begin{array}{l} N_i : s_1 \ \ldots \ s_{j-1} \ \textvisiblespace \ s_{j+1} \ \ldots \ s_{n_i} \\ \{ \\ \quad \textbf{return} \ Node \ (t_1, \ldots, t_m); \\ \} \end{array}$$

ここで $Node$ は部分木 t_1, ..., t_m から構文木を作る関数とする。補完候補計算関数は構文木を受け取って補完候補キーワードの集合を返す関数として構文木の構造に従って相互再帰的に定義する。これらの関数（のいずれか）が $Node \ (t_1, \ldots, t_m)$ を引数に取る場合、キーワード k を一つ含む集合 $\{k\}$ を返すように定義する。

構文規則の右辺に非終端記号がない場合は以下のように生成する。ユーザが与える構文規則において非終端記号 N を左辺に持つ構文規則が以下のように m 個あるとし、m 個すべての構文規則の右辺に非終端記号がないとする。

$$\begin{array}{rcl} N & : & s_{\{1,1\}} \ \cdots \ s_{\{1,n_1\}} \\ & | & \cdots \\ & | & s_{\{m,1\}} \ \cdots \ s_{\{m,n_m\}} \end{array}$$

これら m 個の構文規則はそれぞれ右辺キーワードが 0 個以上あり、m 個の構文規則全体の右辺のキーワードは 1 個以上あるとし、それらを $\{k_1, \ldots, k_i\}$ とする。このとき、m 個の構文規則全体に対し、以下のように字句 ␣ のみを右辺に持つ構文規則を追加する。

$$\begin{array}{rcl} N & : & s_{\{1,1\}} \ \cdots \ s_{\{1,n_1\}} \\ & & \quad \{\textbf{return} \ Empty;\} \\ & & \cdots \\ & | & s_{\{m,1\}} \ \cdots \ s_{\{m,n_m\}} \\ & & \quad \{\textbf{return} \ Empty;\} \\ & | & \textvisiblespace \\ & & \quad \{\textbf{return} \ \{k_1, \ldots, k_i\};\} \end{array}$$

ここで $Empty$ は空の木を表す。追加される ␣ に対し、すべてのキーワードの集合 $\{k_1, \ldots, k_i\}$ を返す。

以上の自動生成アルゴリズムは非常に単純なものであり、実際の言語におけるキーワード補完に適用する際には検討が必要であると考えられる。

6. 入力

ユーザが記述する入力ファイルは 2 つある。一つはユーザが補完したいキーワードを記述するファイルである。C 言語のキーワード `if`、`while`、`void`、`char` などが補完したいキーワードを例として考え、キーワードファイルは以

下のようになる。

```
IF ELSE WHILE VOID CHAR ...
```

もう一つは Yacc の仕様記述ファイルである。C 言語の Yacc の仕様記述を例として考える。まず字句を以下のように宣言する。

```
%token CASE DEFAULT IF ELSE SWITCH WHILE DO
       FOR GOTO CONTINUE BREAK RETURN LP RP
       ...
```

自動生成プログラムはユーザが記述する Yacc の仕様記述において宣言されている字句を読み、キーワードファイルで列挙されたキーワードと照合する。すべての記号を読み終わったら、ユーザに入力として与えられた Yacc の仕様記述の構文規則を読む。C 言語に対する Yacc の仕様記述は以下のようになる。

```
selection_statement
  :  IF LP expression RP statement
  |  SWITCH LP expression RP statement
  ;
  ...
```

7. 実装

5 節で述べた手法に基づき、キーワード補完候補計算プログラムの一部を生成するツールを Byacc[1] のソースコードを利用して実装した。実装したツールのソースコードは http://www.cs.ise.shibaura-it.ac.jp/PRO109/ に公開する。まず 7.1 節において BYacc のソースコードをどのように利用して自動生成プログラムを実装したかについて述べる。次に 7.2 節において字句解析器の実装について述べる。次に 7.3 節で C 言語への適用例について述べる。

7.1 キーワード補完候補計算プログラム生成ツール

実装したツールにおいては、ユーザが補完対象のキーワードを列挙したファイルとユーザあるいはコンパイラ実装者が記述した BYacc の仕様記述ファイルが入力として与えられる。これらの入力をもとに、自動生成ツールは 3 つのファイルを生成する。まず BYacc の仕様記述ファイル (.y ファイル) を生成する。このファイルは、ユーザが記述した BYacc の仕様記述ファイル (.y ファイル) 中の構文規則をもとに生成する。また生成した BYacc の仕様記述中のアクションによって生成する構文木のデータ構造の定義のファイル (.h ファイル)、および構文解析器が生成した構文木を入力として補完候補を計算する関数を定義したファイル（.c ファイル）を生成する。

7.1.1 仕様記述ファイルの自動生成

本ツールでは、ユーザあるいはコンパイラ実装者が記述した BYacc の構文定義ファイル (.y ファイル) をもとに、字句 **error** や入力中のキーワードに対応する字句を含む規則を追加することにより、補完候補計算プログラムで用いる構文解析プログラムを生成するための仕様記述ファイル (.y ファイル) を生成する。

仕様記述ファイル (.y ファイル) の生成は、ユーザが与えた仕様記述ファイルの字句定義と構文規則を読み、字句定義に CURSOR を追加し、構文規則に␣および **error** を含む構文規則を追加し、各構文規則に対するアクションを生成することにより行う。

字句の定義への CURSOR の追加については、入力の BYacc の仕様記述ファイルの%token の部分を読み、ユーザが与えたキーワードファイル内のキーワードと照合できるキーワードがある場合にのみ、生成する Yacc の仕様記述の %token の部分に CURSOR を追加する。

構文規則の追加については、5 節で提示したアルゴリズムに従って、入力の BYacc の仕様記述中の各構文規則の右辺の記号を読み、読んだ記号がユーザが記述したキーワードファイルのキーワードのいずれかと一致する場合、このキーワードについて入力中であることを表す記号と置き換える。

アクションの生成については、入力の BYacc の構文定義ファイル中の各構文規則のアクション記述はすべて削除し、新たにアクションを生成する。アクションは構文木を生成する関数を呼び出す形で生成し、各関数についてプロトタイプ宣言と関数の定義を生成する。自動生成する仕様記述ファイルの各構文規則に対するアクション内で呼び出す関数の名前については、ユーザが与えた BYacc の仕様記述ファイル内の構文規則の左辺の非終端記号の名前を用いることにより、make_exp_1 のように生成する。アクション内で呼び出す関数の実引数については、対応する非終端記号の合成属性を表す$1、$3 などとする。構文規則の右辺に非終端記号がない場合、アクションで呼び出す関数は引数無しの関数とする。

7.1.2 構文木のデータ構造の定義の自動生成

上記で述べたように生成する仕様記述ファイルの各構文規則のアクションで呼び出す関数が生成する構文木のデータ構造の定義を含むヘッダーファイル (.h ファイル) を生成する。構文木のデータ構造は構造体を用いて定義する。各構造体では構文の種類を表す情報を保持する。

7.1.3 補完候補計算関数の自動生成

補完候補を計算する C 言語のプログラムの一部を自動生成する。自動生成する部分については、ユーザが書いた BYacc の仕様記述ファイルに基づいて、上記で生成された構文解析器によって生成される構文木からキーワード補完候補を計算する関数を生成する。これらの関数は各構文の種類毎に相互再帰的に呼び出す形で生成する。

自動生成以外の部分については、Emacs Lisp のプログラムと TCP/IP により通信する関数を書いておく。Emacs Lisp のプログラムは、補完候補を計算する C 言語のプログラムへ現在のバッファの内容を送り、補完候補の計算結果

を受け取ってユーザに提示する機能を実装したものである。

7.2 字句解析器の実装

入力ソースコードの字句解析器は、字句の定義をツールのユーザが記述したファイルを字句解析器生成系に与えることにより生成する。本ツールにおいては、字句解析器生成系として Flex[3] を用いる。ユーザが Yacc の構文記述ファイルと補完したいキーワードを記述したファイルを記述する。Flex の字句記述ファイルもユーザが記述するが、現在の実装においては Lex の字句記述にはカーソルが何らかのキーワードの接頭辞の直後にある状況を表すための␣等の字句の定義をユーザが自動生成によって得られるプログラムと整合性を持つ形で記述することを前提としている。現在の実装ではキーワードが入力中であることを表す字句の定義の生成は行わないため、ユーザが␣を含む字句の定義を与える必要がある。将来的には Lex の仕様記述の一部は対象言語の Lex の仕様記述から自動生成できる可能性がある。

7.3 C 言語への適用例

7.3 節で C 言語の仕様記述ファイルの入力例を挙げて、本ツールを説明する。

7.3.1 仕様記述ファイルの生成

例えば、入力の BYacc の仕様記述ファイルの一部を以下のように記述した場合を考える。

例 1
```
%token CHAR SHORT INT LONG SIGNED UNSIGNED
       FLOAT DOUBLE CONST VOLATILE VOID
%token CASE DEFAULT IF ELSE SWITCH WHILE DO
       FOR GOTO CONTINUE BREAK RETURN
%%
start
    : translation_unit
   ...
selection_statement
    : IF '(' expression ')' statement
    | SWITCH '(' expression ')' statement
    ;
   ...
%%
```

また、補完用のキーワードファイルは以下のように記述してあるとする。

```
IF ELSE WHILE VOID CHAR SHORT
%token CHAR SHORT INT LONG SIGNED UNSIGNED
 FLOAT DOUBLE CONST VOLATILE VOID
%token CASE DEFAULT IF ELSE SWITCH WHILE DO
 FOR GOTO CONTINUE BREAK RETURN LP RP CURSOR
```

字句定義において開き括弧は LP、閉じ括弧は RP と定義している。字句'=' はユーザがキーワードとして指定していないため、補完候補にはならない。

構文規則の左辺に開始記号 start を持つ構文規則については他の構文規則とは別扱いとする。本ツールにおいては、C 言語を対象とする場合、以下のようなアクションを生成する。

```
start : translation_unit
{
  printf ("succeeded.\n");
  parseResult = $1;
}
```

このアクションにより、対象プログラム全体の構文木が変数 parseResult に代入される。

その他の構文規則については、各構文規則に対するアクションは対応する構文の構文木を生成する関数を呼び出す形で生成する。アクション内で呼び出す関数の名前は make_selection_statement_1 のように生成する。引数は、非終端記号の合成属性を表す$1、$3 などとすればよい。構文規則の右辺に非終端記号が無い場合は、引数無しの関数を生成する。例えば、識別子と数の構文規則として以下のような構文規則を考える。

```
primary_expression : IDENTIFIER | CONSTANT
```

この構文規則に対しては、引数無しの関数が以下のように生成される。

```
primary_expression : IDENTIFIER
{
  $$ = make_primary_expression1_1 ();
}
                   | CONSTANT
{
  $$ = make_primary_expression2_1 ();
}
```

構文規則の右辺に非終端記号とキーワードがある場合、キーワードを␣と置き換え、␣の右側の各終端記号を **error** と置き換える。例えば、構文規則の右辺が以下のように記述されているとする。

IF LP *expression* RP *statement* ELSE *statement*

この構文規則の右辺における終端記号をユーザが与えるキーワードと照合し、一致する場合はその終端記号をキーワード入力中であることを示す␣と置き換えて新しい構文規則として追加する。上記の例ではキーワードと照合する終端記号が if と else の 2 つあるので、以下のような構文規則を生成する。

| ␣ '(' *expression* ')' *statement* ELSE *statement*
| IF '(' *expression* ')' *statement* ␣ *statement*

この例において、キーワード else の右側には終端記号がないので、字句 **error** は追加しない。キーワード if の右

側には終端記号があるので、各終端記号を記号 error と置き換えることによって構文規則を生成して追加する。生成した構文規則は以下のようになる。

| IF '(' *expression* ')' *statement* ELSE *statement*
| ␣ '(' *expression* ')' *statement* ELSE *statement*
| ␣ **error** *expression* ')' *statement* ELSE *statement*
| ␣ '(' *expression* **error** *statement* ELSE *statement*
| ␣ '(' *expression* ')' *statement* **error** *statement*
| IF '(' *expression* ')' *statement* ␣ *statement*

この後、各構文規則に対するアクションで呼び出す関数の定義を生成する。各構文規則について、左辺の非終端記号に対して、この構文規則内の非終端記号を用いて、関数の定義を生成する。例として、例 1 のアクションの関数定義は図 1 のように生成される。

本研究で対象にする例において、まず開始記号に対する関数のプロトタイプ宣言を以下のように生成する。

```
struct translation_unit *parseResult;
```

開始記号の構文規則に対する関数のプロトタイプ宣言を生成した後、各構文規則に対する非終端記号に基づいて、例 1 に対する以下のようなプロトタイプ宣言を生成する。

```
struct postfix_expression
  *make_postfix_expression5_1
    (struct primary_expression
      *primary_expression);
```

各終端、非終端記号の属性のデータ型として%union の宣言を生成し、共用体のメンバー名を非終端記号に関連付けるため、%type の宣言を生成する。例えば、例 1 で定義されている非終端記号 selection_statement については、%type の宣言は以下のように生成する。

```
%type <selection_statement> selection_statement
```

手書きの字句解析器が入力中のソースコードを字句解析する時、入力中の文字列の綴りを代入する int_value と *id_value の%union を生成し、各構文規則の左辺の非終端記号についての%union を生成する。例えば、例 1 で定義されている非終端記号 selection_statement を含む%union の宣言は以下のように生成する。

```
%union{
  intint_value;
  char * id_value;
  structselection_statement * selection_statement;
  ...}
```

7.3.2 構文木のデータ構造定義の生成

各構文規則のアクションで呼び出す関数が生成する構文木のデータ構造の定義を含むヘッダーファイルを生成する。まず各アクションで呼び出す関数の定義で扱うタグの宣言を生成する。例 1 のアクションの関数定義で扱うタグの宣言は以下のように生成する。

```
enum tag {selection_statement195_1};
```

各構文木のデータ構造の定義はこの構文規則の左辺の非終端記号に対し、各アクションで生成される構文木の構造体のタグと構文規則の右辺にある全ての非終端記号で構成し、以下のように生成する。

```
struct selection_statement{
 enum tag tag;
 struct expression * expression;
 struct statement * statement;
 struct statement_1 * statement_1;
```

7.3.3 補完候補計算関数の生成

補完候補を計算する C 言語のファイルは手書きで記述するファイルと自動生成するファイルを分ける。main 関数を記述してあるファイルの先頭に以下のようなソースコードを手書きで追加する。ヘッダーファイルの名前を変える場合、ここに手書きで名前を書き直す必要がある。

```
#include "completion.h"
```

関数の宣言はヘッダーファイル (completion.h) の中に生成して追加する。補完候補を計算する関数の名前は文字列 keyword と構文規則の左辺の非終端記号名の組み合わせに指定し、補完候補を計算する C 言語のファイル (completion.c) に追加する。例えば、構文規則の左辺の非終端記号 postfix_expression に対し、生成する関数は以下のように生成する。

```
list keyword_selectioin_statement
  (struct expression_statement * expression_statement,
                          listresult);
```

各補完候補を計算する関数は構文規則左辺の非終端記号と各アクションで生成される構文木の構造体のタグに基づき生成する。例えば、ツールによって生成された以下のような構文規則を考える。

```
␣ LP expression RP statement ELSE statement
```

この例の構文規則のアクション部分で生成される構文木の構造体のタグは SELECTION_STATEMENT193_2 であり、補完候補を計算する関数の一部は以下のようになる。

```
list * keyword_selection_statement
  (struct selection_statement
```

```
struct selection_statement *make_selection_statement195_1 (struct expression *expression,
struct statement* statement){
  struct selection_statement *selection_statement195;
  selection_statement195 = malloc (sizeof (struct selection_statement));
  selection_statement195 -> tag = SELECTION_STATEMENT195_1;
  selection_statement195 -> expression = expression;
  selection_statement195 -> statement = statement;
  return selection_statement195;
}
```

図 1 構文木を生成する関数定義例

Fig. 1 example of the function definition of the generating parsetree

```
    *selection_statement, list result){
  ...
  case SELECTIOIN_STATEMENT193_2:
    return cons("if", NULL);
  ...
}
```

5 節で提示したアルゴリズムにおいては、␣は任意のキーワードを入力途中である状態を示している。上記の例に対し、カーソル位置で期待する補完候補はキーワード `if` であるので、`keyword_selectioin_statement` の返り値はキーワード `if` になる。

8. 考察

本節では自動生成ツールを用いて、実験した結果についての考察を述べる。8.1 節で曖昧な文法を生成する場合について述べる。8.2 節で括弧の補完について述べる。

8.1 曖昧な文法の生成

自動生成機能は Yacc の中置演算子の結合順位の記述（`%left` と`%right`）を考慮していない状況である。自動生成プログラムはユーザが Yacc の仕様記述ファイルに Yacc のキーワード`%left` と`%right` を記述しても、無視する。自動生成プログラムは演算子を含む構文規則がある場合、以下のような警告メッセージを出力する。

```
conflicts: 7 shift/reduce, 90 reduce/reduce
```

このメッセージは 7 個の `shift/reduce` が発生していることと示す。

以下のソースコードを例として考える。

```
int main (void){
  int ii;
  ␣ (ii = 1) ii;}
```

このような入力ソースコードに対応する構文規則は以下の 2 つがある。

```
selection_statement
  :  IF LP expression RP statement
  | ␣ LP expression RP statement
  ...
```

```
iteration_statement
  :  WHILE LP expression RP statement
  | ␣ LP expression RP statement
  ...
```

カーソル位置に 1 文字も入力しない場合、補完候補を計算する際に、同時に提示してほしいキーワードは `if` と `while` であるが、5 節で提示したアルゴリズムでは、キーワード `while` は補完候補にならない。カーソル位置に文字 `w` を入力しても、その部分の字句解析結果は␣である。また、構文規則

```
selection_statement
  | ␣ LP expression RP statement
```

と構文規則

```
iteration_statement
  | ␣ LP expression RP statement
```

の右辺が同一であり、reduce/reduce conflict となり、片方の規則 `iteration_statement` は無視される。そのため、`w` を入力しても `while` は補完対象とはならない。

8.2 括弧の補完

自動生成プログラムが生成したファイルを用いて、補完機能について実験した。まず構文解析する際に、構文誤りがない場合、if 文のキーワード `if` と `else` と開き括弧の補完を行うことができるが、閉じ括弧の補完はできない。例えば、以下のソースコードを考える。

```
int main (void) {
  if (ii ␣ ii;
  else ii;}
```

この補完対象プログラムの if 文に対応する自動生成された構文規則は以下の規則である。

selection_statement : `if` `LP` *expression*
␣ *statement* `ELSE` *statement*

補完候補計算関数のこれに対応する部分は以下のように生成される。

```
case SELECTION_STATEMENT196_9:
```

```
    return cons(")", NULL);
```

閉じ括弧) が補完候補として提示されてほしいが、構文誤りが発生し、構文解析がエラー処理状態で終了し、補完が行われない。␣が先読み文字のとき、shift と reduce の 2 つの遷移があり、shift/reduce conflict となる。デフォルトでは shift が優先であり、閉じ括弧が非終端記号 `expression` から生成される終端記号列の一部として認識されてしまい、期待される構文構造として認識されない。

9. 関連研究

先行研究 [4] において、LR 構文解析の誤り回復を用いた識別子補完方式が提案された。この方式では識別子の補完候補計算プログラムは特別な字句 **error** を含む構文定義をユーザが与える必要があり、また補完計算プログラムは構文解析器以外はユーザが手書きをする必要があった。この方式をキーワード補完に応用し、字句 **error** の構文定義への挿入方式および補完計算プログラムの系統的導出方式を提案し、実装した。自動生成機能は、ユーザが字句定義および字句 **error** を含まない構文定義を与えることにより、補完候補を計算するための Yacc の構文定義、アクション記述、および補完候補計算プログラムを生成する。初心者に対して、補完用のファイルは手書きの手間を減らすという便利な機能を提供する。ユーザに補完用のファイルは手書きの手間を減らすという便利な機能を提供する。

これまでに変数名やキーワードの補完を備えた統合開発環境（IDE）は多く開発されてきた。それらのうちのいくつかは編集中のファイルに入力された文字列に基づいた簡易なキーワード補完機能を備えている。他のファイルの中で定義された識別子を補完対象にする IDE もある。Visual Studio の Intellisense や、Eclipse の content assist[5]、vim の omni completion[6] のようにさらに高度な補完を行うものもある。

また既存研究 [7], [8] では型推論機構を備えた多相型言語の簡易言語に対する変数名補完が提案されている。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、型情報を用いた変数名補完を行う方式である。ただし、カーソル位置より前に構文誤りや型誤りがある場合には補完が行われない。

10. まとめと将来の課題

本論文では、誤り回復を伴う構文解析を行うことにより入力中の（構文が不完全かもしれない）プログラムに対しキーワード補完を行う手法を提案した。さらにユーザの記述仕様から補完候補計算プログラムの一部を自動生成するアルゴリズムを考案し、構文解析器生成系 BYacc のソースコードを修正することにより C 言語で実装した。自動生成アルゴリズムにおいては、構文解析時の誤り回復の制御のため、Yacc の字句 **error** が構文定義中に挿入され、また入力中のキーワードを表す字句が追加される。

構文定義への字句 **error** および入力中のキーワードに対応する字句の挿入方法については、本論文では単純なアルゴリズムを提案し実装したが、この挿入方法について今後検討する必要がある。また現在の実装では Lex(Flex) の仕様のファイルはツールのユーザが手書きをする必要があるが、これを自動生成できるように今後検討を行う。

本研究では、LALR(1) の構文解析器生成系である Yacc を用いることを前提としている。Yacc では shift/reduce conflict が起こった場合、デフォルトで shift を優先し、その場合、適切に閉じ括弧の補完を行うことができない。shift/reduce conflict が起こった場合に reduce を優先することを検討することや、別の方式の構文解析器生成系である ANTLR[9] を用いることを将来試みる。

謝辞

本研究の一部は科学研究費補助金 若手研究 (B) 25730047 の補助を得て行われた。

参考文献

[1] : BYACC — Berkeley Yacc - generate LALR(1) parsers. `http://invisible-island.net/byacc/`.

[2] : Yacc: Yet Another Compiler-Compiler. `http://dinosaur.compilertools.net/yacc/index.html`.

[3] : The Fast Lexical Analyzer. `http://flex.sourceforge.net`.

[4] Sasano, I.: Toward Modular Implementation of Practical Identifier Completion on Incomplete Program Text, *Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies*, BICT '14, ICST, pp. 231–234 (2014).

[5] : Java Content Assist Preferences. `http://help.eclipse.org/juno/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/preferences/java/editor/ref-preferences-content-assist.htm`.

[6] : Omni completion. `http://vim.wikia.com/wiki/Omni_completion`.

[7] Sasano, I. and Goto, T.: An approach to completing variable names for implicitly typed functional languages, *Higher-Order and Symbolic Computation*, Vol. 25, pp. 127–163 (2013).

[8] 後藤拓実，篠埜　功：暗に型付けられた関数型言語に対する変数名補完方式の提案，第 12 回プログラミングおよびプログラミング言語ワークショップ論文集，pp. 216–230 (2011).

[9] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, pp. 425–436 (2011).