Toward Modular Implementation of Practical Identifier Completion on Incomplete Program Text

Isao Sasano
Shibaura Institute of Technology, Tokyo, Japan
sasano@sic.shibaura-it.ac.jp

ABSTRACT

Identifier completion is a widely-used functionality in IDEs like Eclipse and editors like Emacs and vi. In this paper we present how to implement identifier completion for a core of functional languages with a focus on coping with incomplete program text based on error recovery in LR parsing. We believe the present work is a first step toward building practical identifier completion in IDEs for functional languages in modular way by reusing the code in compilers. We also give a specification of the identifier completion and argue that our solution conforms to it.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Program editors*

General Terms

Reliability, Theory, Languages

Keywords

identifier completion, functional languages, lambda calculus, parsing, error recovery, Emacs mode

1. INTRODUCTION

Identifier completion is a widely-used basic functionality in integrated development environments (IDEs) like Eclipse. Editors like Emacs and vi can also be considered IDEs by adding various features depending on languages. Recent IDEs provide identifier completion that takes into account the context surrounding the position of the identifier to be completed. Such completion is called *intellisense* in Visual Studio, *omni completion* in vi, and *content assist* [3] in Eclipse plugins for various languages. We call such completion as *context sensitive* in the rest of the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
BICT 2014, December 01-03, Boston, United States
Copyright © 2015 ICST 978-1-63190-053-2
DOI 10.4108/icst.bict.2014.257909

Major languages like C and Java have IDEs with useful functionalities like identifier completion or various refactorings, but minor languages like Standard ML and Haskell do not at this moment. Developing IDEs takes considerable amount of time and IDEs for minor languages are not expected to be actively developed. In addition programmers may not be attracted to use languages that do not have IDEs with enough functionalities.

In order to remedy such situations, it is desirable to make it easy to develop IDEs. A promising way is to reuse the compiler code for implementing IDEs. Most basic one is lexical analyzer, which is probably used by many of the IDEs. Others presumably include parsing and in some cases type checking when the language is statically typed. Some of the functionalities IDEs provide, like identifier completion, are used while writing programs and others, typically refactorings, are used after writing programs. This paper focuses on identifier completion in the situations where some parts of the program text are incomplete or having mistakes in the lexical and syntactic level.

Developers of IDEs may have their own policy to cope with incomplete program text and may not specify or explicitly explain the functionalities of the IDEs. Programmers may hesitate to use IDEs if the specification of the functionalities are unclear. The proficient programmers would like to know the behavior of IDEs in detail so that they can precisely predict the behavior of IDEs.

In this paper we present an approach to implementing identifier completion with the following features.

- Identifier completion is clearly specified.
- Identifier completion is implemented by reusing the code in compilers as much as possible.
- Behaviors of identifier completion are predictable or controllable by programmers with a certain degree of knowledge about language processing like LR parsing.

The rest of the paper is organized as follows. Section 2 shows our basic ideas and outline of our solution. Section 3 specifies the identifier completion problem. Section 4 describes our implementation as an Emacs mode for a small subset of Standard ML and give some analysis. Section 5 discusses related work. Section 6 describes future work and concludes the paper.

2. BASIC IDEAS

Here we show our idea to cope with incomplete program description in completing identifiers. Note that we do not

care about the type consistency in this paper for simplicity. The idea is simple: to use the functionality of the error recovery in LR parsing, more specifically, Yacc. Although we present the idea in a core of functional languages, it is not essential and the idea can be applied to any other language provided that its syntax is given as an LR grammar.

Yacc [5] is a parser generator for the language C based on LR parsing, more specifically LALR(1), and there are many Yacc-like systems for various languages. Error recovery is supported in Yacc and most Yacc-like systems and the error recovery in Yacc is illustrated in the Yacc web page [5, Section 7] and in a compiler textbook [2, Section 4.9.4].

We briefly review the error recovery in Yacc here. An error is detected when the parser consults the parser table for the current lookahead symbol (token) and the current state and finds no action to do in the corresponding entry in the table. If an error is detected, the parser gets into the error-handling mode and pops its stack until it enters a state where the terminal symbol **error** is legal. It then behaves as if **error** were the current lookahead terminal symbol, i.e., performs the action in the entry for the state and the symbol **error** in the table. The lookahead symbol is then reset to the symbol that caused the error. In order to prevent a cascade of error messages, the parser, after detecting an error, remains in the error-handling mode until three consecutive terminal symbols have been successfully read and shifted. If an error is detected when the parser is already in the error-handling mode, no message is given, and the input terminal symbol is quietly deleted.

In the identifier completion of our approach, the parser generated by Yacc-like system reads the program text being currently edited and constructs a parse tree with some text, like keywords, being added or deleted. Based on the constructed parse tree, our system computes the candidates to be completed with taking into account the scopes of identifiers. The error recovery is controlled by the location where the developers of IDEs insert the special token **error**.

Several ways have been developed to recover from errors in LR parsing and we list four of them here: panic-mode recovery, phrase-level recovery, hand-writing error-handling routines, and error recovery in Yacc. In the present paper we select the fourth way, the error recovery in Yacc, from the point of view of reuse of code in the compiler. Although other selections, especially (partially) hand-writing error-handling routines, may be suitable for practical situations, but we leave them as future work.

3. SPECIFICATION OF IDENTIFIER COMPLETION PROBLEM

In this section we specify the problem that we solve in this paper. We use the following core functional language of static scope, on which we specify the problem.

$$\begin{array}{rcl} M & ::= & x \mid c \mid \lambda x.M \mid M\ M \mid (M) \\ & \mid & \textbf{let}\ x = M\ \textbf{in}\ M\ \textbf{end} \end{array}$$

Here x represents a variable, c represents a constant like an integer, $\lambda x.M$ represents a lambda abstraction, $M\ M$ represents a function application, and **let** $x = M$ **in** M **end** represents a let expression. We have explicitly included the parentheses in the core language in order to express situations where there are open parentheses not yet closed. We excluded type annotations from the syntax for simplicity.

Problem 1 (Variable completion). *Given a (incomplete) term M with a cursor that points at partially input identifiers (including empty spelling) in M, compute candidates to be completed at the cursor position. The candidate identifiers should have as their prefixes the (partial) spelling at the cursor position. If one of them is selected and completed by the users, the obtained term M', possibly with some tokens being deleted or added, constitutes a syntactically legal term that satisfies the scope rule.*

The above problem specification literally implies that any identifier can be a candidate to popup only if it has as its prefix the (partial) spelling at the cursor position, since the specification allows tokens to be freely added or deleted. Since the aim of this paper is to make a first step toward the practical completion system, we make the specification a little loose enough to allow flexible development of the system and in the future may refine the specification to suit practical situations.

4. IMPLEMENTATION

Based on the basic ideas described in Section 2 we have developed an Emacs mode that provides identifier completion for a small subset of the Standard ML [6], which is a functional language of static scope. The source code for the Emacs mode is available on our web page `http://www.cs.ise.shibaura-it.ac.jp/mpse2014/`. The Emacs mode is implemented by using Emacs Lisp and C, where the program in C is a server to compute the candidates and the program in Emacs Lisp sends all the program text in the current buffer and the location of the cursor to the server written in C. It is rather common to write relatively complex computation in languages other than Emacs Lisp. We use Lex and Yacc to generate the program in C.

4.1 Concrete syntax of the core language

As the target language of the implementation we use the following concrete syntax for the core language described in Section 3.

$$\begin{array}{rcl} \mathit{start} & ::= & \mathit{exp} \\ \mathit{exp} & ::= & \mathit{appexp} \\ & \mid & \textbf{fn}\ \mathit{id} \Rightarrow \mathit{exp} \\ \mathit{appexp} & ::= & \mathit{atexp} \\ & \mid & \mathit{appexp}\ \mathit{atexp} \\ \mathit{atexp} & ::= & \mathit{id} \\ & \mid & \mathit{num} \\ & \mid & (\mathit{exp}) \\ & \mid & \textbf{let}\ \mathit{dec}\ \textbf{in}\ \mathit{exp}\ \textbf{end} \\ \mathit{dec} & ::= & \textbf{val}\ \mathit{id} = \mathit{exp} \end{array}$$

The above language is a subset of Standard ML. As for the correspondence with the core language in Section 3, **fn** $\mathit{id} \Rightarrow \mathit{exp}$ corresponds to lambda abstraction $\lambda x.M$ and num corresponds to constant c. In Standard ML the keyword **val** is used for indicating value declarations, in order to distinguish them from function declarations, which we do not use in the present paper.

In the above language identifiers are introduced in the declaration dec and the function **fn** $\mathit{id} \Rightarrow \mathit{exp}$. The scope of the identifier id in **val** $\mathit{id} = \mathit{exp}$ is exp and the scope of the identifier id in **fn** $\mathit{id} \Rightarrow \mathit{exp}$ is exp, with each excluding the scopes of the same id introduced in the respective exp.

4.2 A specification of error recovery

In Yacc, recovering from errors is specified by the locations of inserting the special terminal symbol **error**, which is a reserved word in Yacc to suggest places where errors tend to occur, into the syntax description for Yacc. We show a way of inserting **error** into the syntax description given in Section 4.1. In the following example **error** is inserted into two locations.

$$
\begin{array}{rcll}
\mathit{start} & ::= & \mathit{exp} & (1)\\
\mathit{exp} & ::= & \mathit{appexp} & (2)\\
 & | & \textbf{fn}\ \mathit{id} \Rightarrow \mathit{exp} & (3)\\
\mathit{appexp} & ::= & \mathit{atexp} & (4)\\
 & | & \mathit{appexp}\ \mathit{atexp} & (5)\\
\mathit{atexp} & ::= & \mathit{id} & (6)\\
 & | & \mathit{int} & (7)\\
 & | & (\mathit{exp}) & (8)\\
 & | & \textbf{let}\ \mathit{dec}\ \textbf{in}\ \mathit{exp}\ \textbf{end} & (9)\\
 & | & \textbf{let}\ \mathit{dec}\ \textbf{in}\ \mathit{exp}\ \textbf{error} & (10)\\
\mathit{dec} & ::= & \textbf{val}\ \mathit{id} = \mathit{exp} & (11)\\
 & | & \textbf{error} & (12)
\end{array}
$$

There are various ways to insert **error** to the specification. In the future we would like to develop a systematic way for finding suitable locations for inserting the symbol **error**.

4.3 Action description in Yacc

We use Yacc to generate the parser with error recovery. The output of the parser is a syntax tree with some portion of the original text being deleted or some keywords like `end` being added. For example, for the rule (9) and (10) in the specification in Section 4.2, the action is described as follows.

```
| LET dec IN exp END
{
  $$ = make_atexp4 ($2, $4);
}
| LET dec IN exp error
{
  $$ = make_atexp4 ($2, $4);
}
```

The action description is same for the two rules, which means that the keyword `end`, which corresponds to the token `END`, is virtually inserted if it is missing. Note that the syntax tree is important in our system even if there is any syntax error, while in compilers it does not matter.

4.4 An example

The following is a program fragment in Standard ML, where we inserted the underscore `_` for representing the cursor position. This kind of situation sometimes happens when the programmer starts writing the body of a let expression after writing the keyword `end`.

```
let val xx = 1
in let val xy => 2
   in x_
   end
```

This fragment has a syntax error, where `=>` is the cause of the error. The error would not occur if `=>` were `=`. Another error is that `end` is missing after the `end` in the above fragment, since Standard ML requires `end` for each `let`. In this case one `end` was written in advance and the other `end` will be written later. Generally it depends on programmers in which order the program text is input.

The Emacs mode we have implemented successfully copes with these two errors and shows the candidate `xx` in the popup window. Virtually for the first error the fragment `val xy => 2` is deleted and for the second error `end` is added and thus the obtained candidate `xx` conforms to the specification of PROBLEM 1, although these deletion and addition are not applied to the program text in the current buffer. Figure 1 shows the screen shot of the completion for this example.

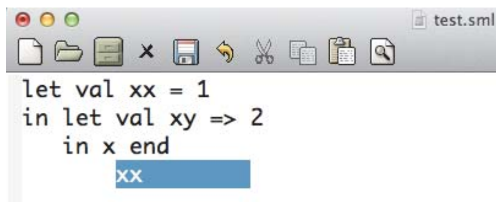

Figure 1: A screen shot of popup in the Emacs mode

4.5 Discussions

Here we analyze how the locations of inserting **error** affect the error recovery. We guess in general there is no way to cover all the possible ways to recover from syntax errors. The following is a program fragment for showing the error recovery does not work in the setting of the specification (1) to (12) in Section 4.2.

```
let val xx = 1
in let val xy => 2
   in x_
```

The fragment is obtained by deleting `end` from the fragment in Section 4.4, so two `end`s are missing. To cope with this situation it may be suitable to use `yyerrok;`, which is a statement that can be used in the action descriptions in the specification of Yacc, to force the parser to get back to its normal mode even if three consecutive terminal symbols are not yet shifted after getting into the error-handling mode.

5. RELATED WORK

There are a few academic work directly concerning identifier completion. In [4, 8] identifier completion for a core of functional languages was presented with considering type consistency. The work considered a simple case with the program text given completely without any syntax errors or type errors up to the cursor position. This is too strict and the present work relaxed the restriction to allow programs having syntax errors in various points, including the text before the cursor position.

Robbes et al. [7] pointed out that finding the candidate from the pop-up window can be slower than typing the full name. They made some assumptions that programmers are likely to use methods they have just defined or modified and that local methods are called more often than the ones in other packages. They claim that the computed candidates include the one the programmer is looking for with high probability. While their approach is based on statistics, our approach is based on error recovery in LR parsing, which means the behavior can be precisely predicted.

Several work about error recovery has been done. One [10] is about *phrase-level* error recovery and *local correction* used in LR parsing in Helsinki Language Processor. When the parser detects an error, the parser tries to find a nonterminal symbol A that is legal by some state q on the stack and an input terminal symbol, not yet read, that is legal at the state goto(q, A). The recovery from errors and local correction may undesirably strongly depend on the description of the grammar. Another is about *panic mode* error recovery. When any error is detected, the parser skips input terminal symbols until some suitable symbol, called *synchronizing token*, is found. The error recovery in Yacc is a kind of mixture of the phrase-level error recovery and the panic mode. In Yacc, the programmer specifies where the error is allowed by inserting the special symbol **error** and the error recovery in Yacc also strongly depends on the location of the symbol **error**.

As for the reuse of compiler code for IDEs, Schäfer et al. [9] presented an implementation of renaming functionality for Java on Eclipse.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented how to cope with incomplete program description in completing identifiers using error recovery in Yacc. Although we described our idea in a core of functional languages, the approach itself can be applied to any other language, provided that its syntax is given in an LR grammar. We believe this is the first work to develop parser-based identifier completion for practical situations where some portions of the program are unfinished. We have developed basic mechanism of identifier completion system with allowing the program having syntax errors typically due to the existence of unfinished portions of the program text. Based on the present work concerning the incompleteness of the program text and the work [4, 8] concerning the type-directed completion, we plan to develop systems for functional languages like Haskell, Standard ML, OCaml, and so on. In order to extend our solution to cover these languages there are several things to overcome, which we leave as future work and describe below.

One is to incorporate the work [4, 8] about type-directed completion into the present work allowing incomplete program description, which is expected to be done modularly.

The identifier completion presented in this paper computes candidates from scratch every time when the programmer types a key, so much redundant computation may be performed. In real program development, a program is divided in many files each of which is not so large. What really matters is the time to process the program in the file currently edited. In particular we can compute necessary information in advance about the identifiers declared in libraries. Moreover in the future we may incrementalize the completion algorithm, which decreases the time to process the file currently edited. There is much work about the reuse of computation and we expect they might be used for the reuse of intermediate results of parsing and type inference. For example, Aditya et. al. [1] proposed an incremental algorithm for type inference. His work enables type inference to be performed in units of the top-level declarations. In the future we plan to develop identifier completion system for real functional languages such as Standard ML, Haskell, OCaml, and so on based on the ideas including ours.

In the error recovery identifiers may be discarded from the stack and thus not included in the resulting parse tree. It would be desirable for such identifiers to be candidates to be completed provided that they suit the context. We will try not to delete identifiers as much as possible.

Although in this paper we focus on identifier completion, we plan to develop context-sensitive completion of expressions, keywords, patterns for syntax, and so on, based on error recovery in LR parsing and type checking.

In the current implementation the specification for Yacc is written by hand since the language was very small. In the future we plan to use the specification for Yacc-like systems in compilers for functional languages like Standard ML.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous referees for many helpful comments. The use of error recovery in the LR parsing to implement identifier completion was presented in Master's thesis of Satoru Suwa [11] under the supervision of the author. This work was partially supported by JSPS KAKENHI Grant Number 25730047.

8. REFERENCES

[1] 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, 1991.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, second edition, 2007.

[3] Content assist. `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors_contentassist.htm`.

[4] Takumi Goto and Isao Sasano. An approach to completing variable names for implicitly typed functional languages. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial Evaluation and Program Manipulation*, pages 131–140, 2012.

[5] Stephen C. Johnson. Yacc: Yet another compiler-compiler. `http://dinosaur.compilertools.net/yacc/index.html`.

[6] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[7] Romain Robbes and Michele Lanza. How program history can improve code completion. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326, 2008.

[8] Isao Sasano and Takumi Goto. An approach to completing variable names for implicitly typed functional languages. *Higher-Order and Symbolic Computation*, pages 1–37, 2013.

[9] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 277–294, 2008.

[10] Seppo Sippu and Eljas Soisalon-Soininen. Practical error recovery in LR parsing. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–184, 1982.

[11] Satoru Suwa. Master's thesis, Shibaura Institute of Technology, Japan, 2013. in Japanese.