Asian Journal of Information Technology 6 (7): 825-833, 2007
ISSN: 1682-3915

Lingu to Java Translator with UUAG Compiler Construction Tool

Heru Suhartanto, Jimmy Bong and Ade Azurat
Faculty of Computer Science, Universitas Indonesia, Kampus UI Depok 16424, Indonesia

Abstract: This study reports our experience in developing Lingu to Java translator based on Attribute Grammar. Lingu is a new specification language to represents reliable system. The specification can be verified by a theorem prover and then transform into Java which will be ready to be compiled and implement. Various compiler construction tools such as UUAG, CUP and JavaCup are studied and used in the development process. The translator manage to transform the Lingu specifications into Java and we identified that the Attribute Grammar constructions tools is the most suitable tool compared to others on almost all of the comparison aspects such as parsing methods, parsing tables, error handling, code structure, modularity, adaptability and the file management

Key words: Lingu, attribute grammar, compiler construction tools, Java translator, UUAG, CUP

INTRODUCTION

There are many compiler construction tools available. The goal is to simplify the process of creating a compiler. Programmers do not need to know the complexity of compiler construction process, such as building a parsing table. The details of how compiler does the job are managed by these tools and usually hidden from the programmers' point of view. The only thing that a programmer needs when building a compiler is basic knowledge of programming and how to represent a language.

The most well known compiler construction tools is YACC (Yet Another Compiler- Compiler) (Johnson, 1979). It is not the first compiler-compiler ever built but it has been a role model for most modern compiler-compilers. Yacc takes a grammar written in Backus-Naur Form (BNF) (Naur, 1963) notation and generates a parser for it. The generated output is a code for the parser in the C programming language. Many variations of Yacc have been developed. CUP (Hudson *et al*., 1999) and JavaCC (Viswanadha and Sankar, 2004) are two examples of compiler-compiler that are similar to Yacc, but use Java programming language instead of C.

Since then, there are many compiler-compiler tools developed. There is a list of some freeware compiler construction tools given by Heng (2005). Utrecht University also builds a compiler-compiler tools which focus on attribute grammar system called UUAG1 (Swiestra and Azero, 1998). Using Haskell programming language, that offers powerful abstraction and typing mechanisms, UUAG is said to bring a new approach of compiler construction. UUAG provides libraries to support compiler construction since attribute grammar is used to particularly define semantics. Parser combinator can be used to assist parser construction. Some functions are also available to help scanning inputs and passing them to the parser. And pretty printing is useful for formatting text-based output as the output of the compiler.

We have experiences in construction MuPL to Java translators using CUP (Suhartanto and Bong, 2006) and Lingu to Java using JavaCC (Suhartanto *et al*., 2006). However, we found some disadvantages using CUP and JavaCC during the processes, for example the process depended on the generation of parsing tables, both CUP and JavaCC will halt after seeing a parsing error hence one needs to manually fix the problem and both have low level of modularity. This motivated us to use UUAG as an alternative approach. In this study, we show the process of building a translator for a language called Lingu into a Java programming language with UUAG and identify how UUAG differs with the other tools.

TRANSLATOR FROM LINGU TO JAVA

The translator that tranforms Lingu into Java programming language is a part of LinguSQL (Suhartanto *et al*., 2006). LinguSQL is an Integrated Development Environment (IDE) for verifying and transforming Lingu scripts. It takes a Lingu script and the associated validation script and reduces them to a set of verification conditions. The validity of these conditions imply the correctness of the main script with respect to its specification. After these conditions are proven valid, LinguSQL will transform the main script into a concrete code.

Corresponding Author: Heru Suhartanto, Faculty of Computer Science, Universitas Indonesia, Kampus UI Depok 16424, Indonesia

LinguSQL consists of 3 different engines. The first is transformation engine for transforming Lingu scripts into concrete codes. The second is verification engine for reducing a scripts specification into a set of mathematical formulas (which are called verification conditions above). These formulas will then be verified using a theorem prover (HOL). And the last is testing engine for executing the test scenarios specified in the given validation script and compare the results to the scripts induced specification. This is done using a dummy database generated by the engine. Finally, the result will be brought to the user representing its validity value.

Lingu is a lightweight language to program data transformations on database (Prasetya *et al.*, 2005). Even though the language is small, Lingu provides enough expressiveness to program a large class of useful data transformations. Lingu embraces validation and testing as an integral part of the programming. This is reflected by integrating validation scripts inside Lingu. But Lingu is unexecutable. Therefore, a Lingu script needs to be translated into a concrete executable program. To translate Lingu scripts into programming languages, UUAG is taken as an approach to build the translator. Java is the initial choice for the target language. In the next section, the construction process is partially shown. The details are discussed in Jimmy (2006).

DEFINING DATA STRUCTURE IN UUAG

In UUAG, the keyword DATA can be used to help defining the data structure that keeps the significant information from the input file. When it is compiled using UUAG compiler, it will generate a data-type definition in Haskell. To initially define the data structure, the grammar can be used as a basis. Each production in the grammar is given a definition for a certain data type. Below is given an example of a single production rule to form Lingu program.

Program →TypeDeclaration * ClassDeclaration+ (1)

From the production, we know that a program consists of a number of type declarations and one or more class declarations. The information gives us a lead on how to describe the data structure to store the information of a Lingu programwhich is as follows:

```
DATA Program
    | Program typedecls : TypeDeclList
      classes : ClassList
```

The code above gives us a definition of Lingu program. It states that a Lingu program has two components: Typedecls and classes, both represents a group of type declarations and a group of class declarations, respectively. The type TypeDeclList and ClassList are the types that we will yet define, but it is common to use Haskell list to contain the multiple elements.

UUAG provides TYPE keyword to make aliases for types. Sometimes it is more convenient to give aliases to certain complex types for the sake of clarity. For the implementation of the translator, we use TYPE for naming types of lists. TypeDeclList and ClassList are the examples of types defined using aliasing. Both types are the aliases for [TypeDecl] and [Class], respectively.

```
TYPE TypeDeclList = [TypeDecl]
TYPE ClassList    = [Class]
```

After the data structures have been constructed for every aspect of the Lingu language, the next step is to define the parser.

BUILDING THE LINGU PARSER

We use parser combinator to build the parser. It is provided in the UUAG package and easier to use rather than to program the parser from scratch using Haskell programming. It is a quite powerful tool to build a parser in a Haskell environment which allows parser composition. Parser combinator has a restriction. It does not allow left-recursion in the grammar's productions. Even though it can take ambiguous grammars but such grammars should be avoided for the sake of clarity and efficient parsing. These restrictions are exactly similar to LL grammars. Using (1), we show a parser definition for Lingu program as a conjugation of 2 smaller parser, which are the parser for a group of type declarations and another group of classes definitions.

```
pProgram :: Parser Token Program
pProgram = Program_Program
             <$> pTypeDeclList
             <*> pClassList
```

The parser pProgram uses 2 parsers for its definition. It does not need to define specifically of how it should parse type declarations and Lingu classes. pProgram only states that a Lingu program is mainly composed by 2 major components, i.e., type declarations

and classes in specific order. How the Lingu type declaration and class are structured is handled by more specific parsers, in this case: pTypeDeclList and pClassList. Program_Program is the constructor for Program datatype. It is generated from the data structure that has been defined previously using UUAG. Program_Program takes two arguments that will be resulted from pTypeDeclList and pClassList as Program has been declared to have two components of the types TypeDeclList and ClassList, respectively.

The parser definitions for TypeDeclList and ClassList are given as follows:

```
pTypeDeclList :: Parser Token TypeDeclList
pTypeDeclList = pList pTypeDecl

pClassList :: Parser Token ClassList
pClassList = pList1 pClass
```

Both definitions above take a collection of components and store them in Haskell list. pList and pList1 denote that the parser can take consecutive components, where pList allows no such component to occur while pList1 insists on having at least one occurrence of the expected component. PTypeDecl is a parser for a single Lingu type declaration and pClass is for a single Lingu class.

```
pTypeDecl :: Parser Token TypeDecl
pTypeDecl = TypeDecl_TypeDecl
          <$ pKey "type"
          <*> pConid
          <* pSpec '='
          <*> pTyTypeDecl
pClass :: Parser Token Class

pClass =  Class_Class
          <$ pKey "class"
          <*> pConid
          <*> pParens_pCommas pParam
          <* pOCurly
          <*> pList pMethod
          <*> pList pValidation
          <* pCCurly
```

Apart from the ones used in the definitions above, there are other library functions of parser combinator which will not be discussed here. The definitions are presented to show how we can compose a big/complex parser from some smaller ones, just as we write it in BNF-style. After the parser for Lingu is completed, we define the semantics to process the information acquired from the parsing.

DEFINING SEMANTICS IN UUAG

The keyword SEM is used to define semantics in UUAG. Haskell programming language is used to describe the details of the semantic actions. In building a translator, the final result is another programming code. To help the formatting of the output, pretty printing library in UUAG is used. It provides some functions to simplify textual formatting. But before that, the attributes must be declared first. Since UUAG uses the attribute grammar concept (Knuth, 1968), semantic meanings are defined using attributes. The keyword ATTR declares attributes for each node that we defined previously as data types.

For example, consider the code fragment below.

```
ATTR Program [ | | lg : PP_Doc ]
ATTR TypeDecl [ | | lg : PP_Doc ]
ATTR Class [ | | lg : PP_Doc ]
ATTR TypeDeclList [ | | lgs: PP_Docs ]
ATTR ClassList [ | | lgs: PP_Docs ]

SEM Program
        | Program lhs.lg = (vlist. insert_vsep "" $ @type
        decls.lgs)
                >-< text ""
                >-< (vlist. insert_vsep "" $ @classes.lgs)

SEM TypeDecl
        | TypeDecl lhs.lg = text "public class"
                >#< text @name
                >-< pp_braces @type.lg

SEM Class
        | Class lhs.lg = text "public class"
                >#< text @name
                >-< pp_braces ( dbm_decl
                   >-< dbm_init @name
                   >-< vlist @methods.lgs )
{
dbm_decl = text "private DBManager dbm;"

dbm_init name = text "public"
        >#< text name
        >|< text "()"
        >-< pp_braces ( text "this.dbm = new DBManager
        (\"lingusql\");"
                >-< text "this.dbm.connect();" )
}

SEM TypeDeclList
        | Cons lhs.lgs = @hd.lg : @tl.lgs
        | Nil lhs.lgs = []
```

```
SEM ClassList
        | Cons lhs.lgs = @hd.lg : @tl.lgs
        | Nil lhs.lgs = []
```

The attributes lg and lgs are variables to store the semantic results. They are defined as synthesized attributes, which means their values are calculated from their children's attributes. The type PP_Doc is the type used for pretty printing. The attribute lg of Program composes the values from the attributes of its children: typedecls.lgs and classes.lgs. The functions and operators that are used to manipulate the values are available in the UUAG's pretty printing library. For the types of Haskell lists (e.g TypeDeclList and ClassList), eachprovides two kinds of constructors, i.e. Cons for non-empty list that has two elements hd (head) and tl (tail) and Nil for empty list. As the top most node, the attribute of Program will give the final translation result. The PP_Doc variable is printed out to a file and gives the Java source code

Figure 1 describs the main translation scheme, a lingu file named "berkas.lingu" will be transformed into its Java codes. The following Code1 example shows an example of Lingu code which is translated into Java Code 2.

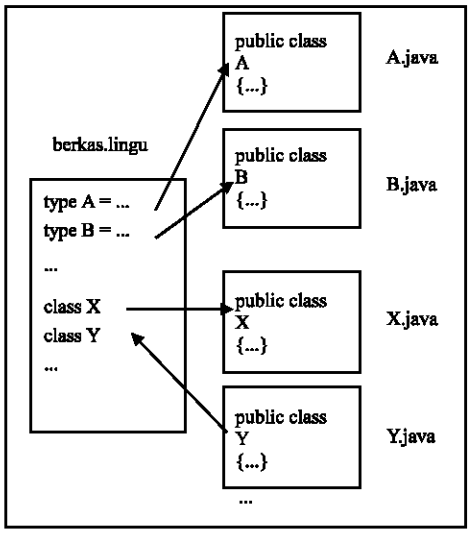

Fig. 1: Main translation scheme

Code 1. Example of Lingu file

```
Type registration table = Record
    {| ID              :: String;
       Name            :: String;
       Sex             :: Integer;
       Category        :: Integer;
       Study programme :: String; |}

Type answer form table = Record
    {| ID              :: String;
       Name            :: String;
       SheetCode       :: String;
       Answer          :: String; |}

type SETdb = Dbase
    {| SubmitTab              :: Table AnswerFormTable;
       MasterTab              :: Table RegistrationTable;
       UnknownAFormTab        :: Table AnswerFormTable;
       DoubleAFormTab         :: Table AnswerFormTable; |}

class SETutility (d::SETdb) {

       method filterUnknown(SubmitTab :: Table AnswerFormTable,
                  MasterTab            :: Table RegistrationTable,
                  UnknownAFormTab      :: Table AnswerFormTable
                ) :: ()
 ids, okids :: Table {| ID :: String; |};
 do
 {
       ids := findAll s<-d.SubmitTab where T found s.ID,s.Name;
       insertAll i<-ids,r<-d.MasterTab where i.ID==r.ID to okids;
       delete ids where ids.ID in okids.ID;
       insertAlls<-d.SubmitTab, i <-ids where s.ID==i.ID to d.UnknownAFormTab;
  }

 method filterDouble(i :: Integer) :: ()
 do
 {
```

Continue: Code 1

```
    ids := findAll s<-d.SubmitTab, r<-d.SubmitTab
          where s.ID==r.ID /\ s <> r found s.ID;
    insertAll s<-d.SubmitTab, i<-ids where s.ID==i.ID to d.DoubleAFormTab;
  }
}
```

Code 2: Translation result of Code 1

Answer form table.java:

```
public class AnswerFormTable
{private String ID ;
 public void setID(String ID)
 {
 this.ID = ID;
 }
 public String getID()
 {
 return this.ID;
 }
 private String Name ;
 public void setName(String Name)
 {
 this.Name = Name;
 }
 public String getName()
 {
 return this.Name;
 }
 private String SheetCode ;
 public void setSheetCode(String SheetCode)
 {
 this.SheetCode = SheetCode;
 }
 public String getSheetCode()
 {
 return this.SheetCode;
 }
 private String Answer ;
 public void setAnswer(String Answer)
 {
 this.Answer = Answer;
 }
 public String getAnswer()
 {
 return this.Answer;
 }}
```

RegistrationTable.java:

```
public class RegistrationTable
{private String ID ;
 public void setID(String ID)
 {
 this.ID = ID;
 }
 public String getID()
 {
 return this.ID;
 }
 private String Name ;
 public void setName(String Name)
 {
 this.Name = Name;
 }
 public String getName()
 {
 return this.Name;
 }
```

Continue: Code 2

```
private Integer Sex ;
public void setSex(Integer Sex)
{
this.Sex = Sex;
}
public Integer getSex()
{
return this.Sex;
}
private Integer Category ;
public void setCategory(Integer Category)
{
this.Category = Category;
}
public Integer getCategory()
{
return this.Category;
}
private String StudyProgramme ;
public void setStudyProgramme(String StudyProgramme)
{
this.StudyProgramme = StudyProgramme;
}
public String getStudyProgramme()
{
return this.StudyProgramme;
}}
```

SETdb.java:

```
public class SETdb
{private AnswerFormTable[] SubmitTab ;
public void setSubmitTab(AnswerFormTable[] SubmitTab)
{
this.SubmitTab = SubmitTab;
}
public AnswerFormTable[] getSubmitTab()
{
return this.SubmitTab;
}
private RegistrationTable[] MasterTab ;
public void setMasterTab(RegistrationTable[] MasterTab)
{
this.MasterTab = MasterTab;
}
public RegistrationTable[] getMasterTab()
{
return this.MasterTab;
}
private AnswerFormTable[] UnknownAFormTab ;
public void setUnknownAFormTab(AnswerFormTable[] UnknownAFormTab)
{
this.UnknownAFormTab = UnknownAFormTab;
}
public AnswerFormTable[] getUnknownAFormTab()
{
return this.UnknownAFormTab;
}
private AnswerFormTable[] DoubleAFormTab ;
public void setDoubleAFormTab(AnswerFormTable[] DoubleAFormTab)
{
this.DoubleAFormTab = DoubleAFormTab;
}
public AnswerFormTable[] getDoubleAFormTab()
{
return this.DoubleAFormTab;
}}
```

Continue: Code 2

SETdb.java:

```
public class SETutility
{private DBManager dbm;
 public SETutility()
 {this.dbm = new DBManager("lingusql");
 this.dbm.connect();}
 public void filterUnknown(AnswerFormTable[] SubmitTab,RegistrationTable[] MasterTab,AnswerFormTable[] UnknownAFormTab)
 {VirtTable vtables = new VirtTable();
 vtables.clearColumnList();
 vtables.addColumn("ID","String");
 vtables.create("ids");
 vtables.clearColumnList();
 vtables.addColumn("ID","String");
 vtables.create("okids");
 this.dbm.findAll("SubmitTab as s", "ids", "T", "s.ID,s.Name");
 this.dbm.insertAll("ids as i,MasterTab as r", "okids", "i.ID = r.ID");
 this.dbm.delete("ids", "ids.ID in (select ID from okids)");
 this.dbm.insertAll("SubmitTab as s,ids as i", "UnknownAFormTab", "s.ID = i.ID");
 }
 public void filterDouble(Integer i,AnswerFormTable[] SubmitTab)
 {VirtTable vtables = new VirtTable();
 this.dbm.findAll("SubmitTab as s,SubmitTab as r", "ids", "s.ID = r.ID and s <> r", "s.ID");
 this.dbm.insertAll("SubmitTab as s,ids as i", "DoubleAFormTab", "s.ID = i.ID");
 }}
```

UUAG COMPARISON TOWARDS CUP AND JAVACC

The comparison of UUAG against CUP and JavaCC is based on experience in using the tools in practice. We will see how UUAG stands out from most compiler-compiler tools.

Parsing method: Parser combinator takes the input and matches token by token to the grammar. It works like an recursive-descent parser (Baars *et al*., 2004) although it does not reject ambiguous grammar. In most cases, ambiguous grammar is not desirable in the sense that programming languages in particular should be rigorous. CUP is similar to Yacc. It generates LALR parser. On the other hand, JavaCC generates LL parser.

Parsing table: There is no generation of parsing table in parser combinator. The parser is built entirely in Haskell. Productions are defined as functions and each production acts as a parser itself. Small parsers are joined together with parser combinatory to create bigger and more complex parsers. Meanwhile, CUP and JavaCC produce parsing tables. Each has its own syntax rules about defining the grammars. The file containing the grammar definition is then compiled to generate parser codes that contain the parsing table for the grammar.

Error handling: Parser combinator takes each error and tries to suggest a fix for the error. It is done by deleting or inserting a token based on the list of token that the parser was supposed to accept. This makes it possible for the parser to finish parsing and inform the user about the location of the errors and its repair solutions. But the parser is naive and can cause larger number of errors as it is trying to fix the errors. If the parser mistreated an error, it would invoke a list of new errors and would run out of resource in the continuous effort of repairing error by error.

By default, CUP will halt if it encounters an error when parsing. But, CUP has error recovery mechanisms, which is similar to standard Yacc error recovery. The recovery uses an error production. This production can be used to reduces erroneous input.

In JavaCC, parser halts when it comes to an error. As all LL parser is a predictive parser, it gives a list of tokens that it should take instead of the one that caused the error.

Code structure: UUAG provides complete library to create a compiler. Focusing mainly on the semantic analysis, UUAG gives the tools to construct the scanner, parser and pretty printer. The libraries can be optionally used, depending on how the user needs. It is possible to define the scanner and parser in Haskell by oneself without using the libraries provided, but still use UUAG to deal with semantics.

The coding in UUAG is very flexible. One can code the whole compiler in a single UUAG file, or one can make it as modular as possible. Scanner, parser and the semantic actions are defined using Haskell programming language. While the semantic definitions in attribute grammar follows the UUAG structure, which will be compiled to produce Haskell codes.

CUP and JavaCC have stricter rules for the codes. Both are based on Java programming language and have a certain format to define parsers. Each needs to be compiled and outputs Java files.

CUP provides a template to build parsers. Grammars are defined in a BNF style and the tokens are defined with regular expressions. For the scanners, CUP uses support from scanner generator tools, such as JLex or JFlex. Semantic actions are embedded in the grammar definition. For more complex semantics, it is possible to create external Java codes that are linked from the semantic actions defined in the grammar.

In JavaCC, users build parser and scanner in the same file. The semantics can be defined in the files generated by JavaCC in a Java-style. It makes JavaCC to be more rigid about the format rules.

Modularity: UUAG gives the freedom to make a compiler as modular as possible. The parser can be built from a composition of smaller parsers to make a bigger and more complex one. The definition of the attributes can also be defined modularly.

Building a compiler using CUP, mainly consists of two major components: the scanner and the parser. CUP handles the parser generation; while the scanner is built using Jlex or Jflex. Defining semantic actions outside of CUP is optional. As a feature of Java programming language, users can create a modularized semantic definitions.

JavaCC has strict rules upon its usage. Users must follow their platform in defining all the components. The scanner and parser are both defined in a JavaCC file. When a JavaCC file is compiled, it will generate one file for each non terminal where the semantic action can be written. In such way, the modularity of the parser is mainly controlled by JavaCC.

Adaptability: Since UUAG supports high-level of modularity, it implies a high-support of adaptability. For instance, when the grammar needs an extended structure, it is possible to define the structure individually and then merge it with the original grammar. The merging can be done easily just by linking them together with a minimal change to the old structure.

Applying changes in CUP grammar would be quite difficult if the grammar is very big. Since the grammar is written in a single file, one should browse all over the grammar to identify the section to be changed. It is a different case if the changes only involve the semantics where they are defined in separate files.

Table 1: Summary of UUAG comparison towards CUP and JavaCC

Comparison aspects	Tool	Description
Parsoing method	UUAG	Recrusive-descent
	CUP	LA LR
	JavaCC	LL
Parsing table	UUAG	None
	CUP	Created upon CUP compilation
	JavaCC	Created upon JavaCC compilation
Error handling	UUAG	Attempt correction
	CUP	Error token mechnism
	JavaCC	Report the token expected
Code structure	UUAG	UUAG attributes for semanties, optional parser and scanner
	CUP	CUP for parser, Jlex/JFlex for scanner, samanties in Java
	JavaCC	JavaCC for parser and Scanner, semantic files generated
Modularity	UUAG	User privalge
	CUP	Upon using Java programming for semantics
	JavaCC	Determined by JavaCC
Adaptability	UUAG	Relativity easy
	CUP	Hard for large grammers
	JavaCC	Relativity hard
File management	UUAG	User decision
	CUP	Sementics can be defined separately
	JavaCC	Determined by JavaCC

JavaCC generates its semantic files whenever it is compiled. The semantic files should be carefully managed, otherwise they would be overwritten by new files in every compilation of the JavaCC file. JavaCC also has the same problem with CUP in the case of applying changes for a large grammar (Table 1).

File management: With UUAG flexibility, the user is given the freedom to arrange how the compiler will be coded. UUAG does not give any restriction whatsoever. The user should decide whether the codes will be limited into a few large files of probably many small files. Of course, the decision should be based on the complexity level of file management for maintenance and further development.

CUP guides the creation of the parser and scanner. As for support features, CUP gives the freedom to extend the parser using Java programming language. Therefore, other than the parser and scanner codes, the programmer should manage the additional files.

File management in JavaCC is mainly done by itself. With strict rules about the usage, JavaCC dictates a programmer how to define a compiler. It even generates the semantic files. Even though it is still possible for programmer to define some additional files to extra support such as more complex semantic functions, but JavaCC has handled mostly everything.

CONCLUSION

Lingu to Java translator is developed using UUAG compiler construction tool. UUAG is a powerful compiler

construction tool. It combines the simplicity of defining semantics using the concept of attribute grammar and the strength of Haskell programming language concerning lazy evaluation and higher order functions. The other advantage of UUAG is how it supports modularity that impacts on a lenient way to deal with changes.

In spite of the advantages that UUAG has to offer, defining a grammar for UUAG is a task that should be carefully done. Mistakenly defined grammar can result the program to go into an infinite loop until it runs out of resource. Its indulgent characteristic towards ambiguous grammar can bring worse problems as it can confuse the interpretation of a language.

Every tool has its pluses and minuses, including compiler construction tools like UUAG, CUP and JavaCC. But the distinct features that each tool has to offer can give the options for the programmer to pick one that is most suitable with the requirements of the problem. Table 1 summarizes the tools comparison result observed.

REFERENCES

Aho, A.V., R. Sethi and J.D. Ullman, 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley.

Baars, A., A. Dijkstra, J. Hage, B. Heeren, B., A.L¨oh, P. van Oostrum, D. Swierstra and W. Swierstra, 2004. Implementation of Programming Languages. Lecture Notes, Utrecht University.

Heng, C., 2005, Free Compiler Construction Tools: Lexers, Parser Generators, Optimizers., http://www.thefree country.com/programming/compilerconstruction.s html.

Hudson, S.E., F. Flannery and Ananian, 1999. CUP LALR Parser Generator for JavaTM. http://www2.cs.tum. edu/projects/cup/.

Jimmy, 2006. Pengembangan Penerjemah Lingu ke Java dengan Attribute Grammar. Master Thesis. Faculty of Computer Science, University of Indonesia.

Johnson, S.C., 1979. YACC: Yet another compiler-compiler. Unix Programmer's Manual, Vol. 2b.

Knuth, D.E., 1968,Semantics of Context-Free Languages. Mathematical Systems Theory, 2: 127-146.

Naur, P., 1963. Revised Report on the Algorithmic Language Algol 60.

Prasetya, I.S.W.B., A. Azurat, T.E.J. Vos, A. van Leeuwen and H. Suhartanto, 2005, Theorem Prover Supported Logics for Small Imperative Languages. Institute of Information and Computing Sciences, Utrecht University.

Suhartanto, H. and Bong, Jimmy, 2006. Penerjemah MuPL ke Java dengan CUP, Copy rights certificate number 029004.

Suhartanto, H., R. Wenang, I.S.W.B. Prasetya, B. Wibowo, S. Maizir and A. Azurat, 2006. LinguSQL: A Verification and Transformation Tool for Database Application. Copy rights certificate number 032110.

Swierstra, S.D. and P.R. Azero, 1998. Attribute Grammars in the Functional Style. Proceedings of the Systems Implementation 2000. Chapman-Hall, pp: 180-193.

Viswanadha, S. and S. Sankar, 2004. Java Compiler CompilerTM(JavaCCTM)- The Java Parser Generator, https://javacc.dev.java.net/.