MetaCET: An Object Oriented Tool for Language Design

Francisco Gortázar, Micael Gallego, Abraham Duarte*
Universidad Rey Juan Carlos
{francisco.gortazar, micael.gallego, abraham.duarte}@urjc.es

Abstract

Programming Language design is one of the most difficult tasks in software design. Nowadays, is an open research area, with a great impact on the further development of the Computer Science. In this paper, we present MetaCET, a new approach for language design. It allows developers to model and design a target language by means of Java.

When modeling, the first step consists of giving a high level language design by a diagram, for instance, UML. Then, in MetaCET, the model is rewritten into Java classes. Inside these Java classes the syntax (or target language textual representation) is specified as metadata (standard Java annotations). Finally, MetaCET automatically generates a Java code which recognizes a program written in the target language.

1. Introduction

Language specification or language design has been a main research area since ALGOL 60 was designed [1]. A number of formalisms or specifications for language design have been proposed, some of them are: Extended Backus Naur Form (EBNF, ISO 14977 standard), ASDL [2] or SDF2 [3]. However, designing a new language is still a hard task. This difficulty has been pointed out by a wide number of authors [4, 5, 6]. Some reasons are given in [7] by Kutter, being the most important one the cost of designing and implementing the language. Among other reasons, it may be caused by the lack of Integrated Development Environments (IDEs) for language design or because specific tools must be used.

The use of general programming languages (GPLs) for language design can make easy the work of language designers. By means of using a GPL, language designers can be benefited both from powerful IDEs commonly available and language knowledge.

In this paper MetaCET is introduced as a tool for language design. It takes as input a target language specified by Java classes with annotations (which establish the target language syntax). The output of MetaCET is Java code which recognizes whatever program written in the target language (the target program) and builds a tree-like representation of the target program. Notice that this tree representation is similar to a Document Object Model (DOM) tree for an XML file. This tree is usually known as an Abstract Syntax Tree (AST) in compiler construction. The generated tree is useful for several tasks like inspection, code generation or validation, among others.

The rest of the paper is organized as follows: previous approaches are reviewed in Section 2, the third Section presents MetaCET's approach and a case study is introduced, in Section 4 a discussion is presented, finally, the paper is concluded in Section 5.

2. Related work

In the literature there are two main approaches for language designing. The first one, named *grammar-based*, uses a grammar as the way for language designing. The second one, called *high level language-based*, focuses on using high level languages for the same task. Obviously, each approach has several advantages and disadvantages (see [8, 9]).

Our proposal is placed in the second approach. For this reason, we only revise the most relevant tools in the second approach.

For the evaluation of the different systems, the following features have been analyzed:

- How is the target language modeled?
- How is the target language syntax specified?

* This work has been partially supported by MCyT TIN2005-08943-C02-02

0-7695-2629-2/06/$20.00 (c) 2006 IEEE

- What are semantics and ease of using of the target program representation?

Some examples of high level language design tools are Meta Programming System (MPS)[1], XMF-Mosaic[2], and JastAdd II [10]. Table 1 presents a comparison of these tools in terms of the previous features. These tools use object oriented languages for language design. The Meta Programming System has its own object oriented language. This language is used in MPS to specify the target language model. The target language syntax is specified using a different language, specifically tailored for this task. The target program representation is accessed by means of another different language. In XMF-Mosaic the target language model is a MOF model [11]. The target language syntax is specified in XBNF, an extension of the BNF formalism. The target program representation is accessed by means of XOCL, an extension of the OCL language which makes it executable. In JastAdd the target language model is specified by means of Java, and the syntax is specified using another, independent, tool. The target program representation is managed through aspects in the original language model.

Table 1. Tools for high level language design.

Tool	Model	Syntax	Semantics
MPS	Specific	Specific	OO language
XMF-Mosaic	MOF	XBNF	OCL, XOCL
JastAdd II	Java	External tool	AOP, Java

It is a common practice that semantic actions are defined by means of GPLs such Java. That is the case of MPS and JastAdd. The main advantage of MPS and XMF are their IDEs. However, both of them require knowledge of different languages for each analyzed feature. XMF is easier (from a high level point of view) than others since it uses high level languages already used in other software engineering tasks, such OCL and MOF.

3. MetaCET's approach

MetaCET is an object oriented Java tool for language design. The main idea of our proposal is modeling a target language, from the beginning to its final stage, by means of Java. It takes as input a target language specified by Java classes with standard-Java annotations (which establish the target language syntax). The output of MetaCET is Java code which recognizes whatever program written in the target language and builds a tree-like representation of the target program.

The main difference regarding to other high level language design tools is how the target language and target programs are managed. In MetaCET all the phases of language design are performed with Java. Our approach tries to solve the language design problem by using tools which are well-known by programmers. For this purpose, we propose using the GPL Java, which enables working in IDE (and other related tools). Furthermore, IDEs and their facilities (syntax highlighting, code completion or refactoring) can be used when coding the classes. As it was explained in section 2, these facilities are not generally available when using other tools for language design.

The rest of this Section is structured as follows: first subsection explains how to design a language with MetaCET; second subsection introduces a case study; and third subsection presents the whole architecture of MetaCET.

3.1. Language Design with MetaCET

In high level language-based approach language design is intended to be performed in three steps. The first one consists of defining the language model. The most common language aimed at this task is UML. The second one consists of rewriting the model into the required format of each language design tool. Generally, in this step the lexical and syntactical features are defined. Finally, each tool generates code with lexical, syntactical and semantic information, which would allow executing a target program written in target language.

In MetaCET the first stage is similar to the other tools. However, as long as MetaCET requires Java code, if UML is used to model the target language, then transformation into MetaCET's required-format can be performed automatically by an UML tool.

In the second stage the model has to be rewritten into Java classes (although, this task may be done automatically, as was explained above). This is the required format for MetaCET. Lexical and syntactic information of the target language is given at the end of this stage. Both are defined as class and method annotations (standard-Java annotations as defined by Java Specification Request-JSR175).

In the final stage, MetaCET generates a parser from annotated Java classes. The parser receives as input a target program and produces as output the target program representation, which is an object oriented tree. Objects in the tree represent each part of the

[1] JetBrains, http://www.jetbrains.com/mps/
[2] Xactium, http://albini.xactium.com/content/

0-7695-2629-2/06/$20.00 (c) 2006 IEEE

source code and they are instances of the classes used to model the language. Information about source code can be obtained invoking methods on these objects.

3.2. Case study

In order to clarify ideas presented in our work, a simple example is studied (extracted from the literature [7]). The language is a **`goto`** language. It consists of a list of statements of three different kinds:

- **`goto`** statements;
- **`print`** statements;
- **`labeled`** statements.

Figure 1 shows an example program written in the **`goto`** language. This program prints/displays an infinite sequence of 0 and 1.

```
start : print 0
print 1
goto start
```

Figure 1. A sample program in the **`goto`** language.

As it was exposed before, the first step consists of a high level design. In this case, a UML model has been proposed. In figure 2 the model is presented.

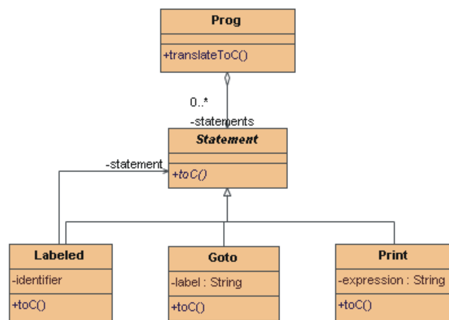

Figure 2. **`goto`** language UML model.

Every program written in the target language (**`goto`** language) is an instance of this model.

The second step consists of rewriting the model into MetaCET format. In this case, from the UML model, throughout an UML tool, Java classes are automatically generated, where each class represents a language concept. For this example, five files are generated, each one containing one class.

In Figures 3-a and 3-b classes defining the **`goto`** language are presented. These classes contain additional information about the language described throughout the next paragraphs.

Classes properties are specified using methods that adhere to JavaBeans[3] style. The name must begin with **`get`** followed by the property name. For instance, the method **`getLabel()`** in **`Labeled`** class (line 10) is a read method for the label property.

Lexical and syntactical features are defined with Java annotations. In **`Goto`**, **`Print`** and **`Labeled`** classes, lexical and syntactical features of each statement are respectively defined. For instance, in **`Print`** class, with **`@Syntax`** is fixed the token which represents the **`print`** statement (line 4). Notice that it is very easy changing whatever keyword. For example, **`print`** with **`write`** in the concerned interface.

The **`@Syntax`** annotation contains the syntactic description for that element of the language. For a labeled statement its value is **`"label ':' labeledStatement"`**. This definition consists of a sequence of property names and literals. Literals are enclosed in single quotes.

The **`Statement`** class is just a super-class of previous ones for polymorphic purposes. In other words, a program is a list of statements (independently of their types). As a consequence **`@Syntax`** is empty.

Finally, the **`Prog`** class is the main one and defines the statement list syntax. Moreover, this class may be annotated with lexical definitions (**`@Tokens`**, lines 5-8) which could be used in other classes. For example, the **`start`** label is defined in this class as an identifier (in line 6: a letter followed by zero or more letters and digits) and used in **`labeled`** and **`goto`** statements (lines 9 and 8, respectively). **`@Tokens`** contains a list of token definitions (**`@TokenDef`** annotations), where each one has a name and a lexical description in JFlex[4] format.

List properties in the model can be annotated with **`@SyntaxList`** (see line 14 of Prog.java). The attribute **`separator`** of this annotation defines the separator for the elements in the list. In this example, statements are separated by a semicolon (but it could be changed by whatever symbol).

Finally, the third step generates a parser. This parser builds an object oriented representation of the target program. This representation is a tree of objects. Nodes (objects) in the tree are instances of the classes.

As the language is designed with Java classes it is straightforward to add semantic actions to the model with standard Java methods. In figure 3 semantic actions for translating a **`goto`** program into C code is presented. The **`Prog`** class defines a method called **`translateToC()`** which constructs a string by

[3] http://java.sun.com/products/javabeans/docs/spec.html
[4] JFlex- The Fast Scanner Generator for Java. http://jflex.de/

calling iteratively the **toC()** method for each **Statement** (lines 18-26). Notice that polymorphism is used for this task.

Prog.java

```
1 package gotolanguage;
2 import java.util.List;
3 import es.urjc.escet.metacet.annotation.*;
4
5 @Tokens({
6   @TokenDef(name="ID", lex="[:jletter:][:jletterdigit:]*"),
7   @TokenDef(name="CTE", lex="[:digit:]*")
8 })
9
10 @Syntax( " statements " )
11 public class Prog {
12   private List<Statement> statements;
13
14   @SyntaxList(separator=" ';' ")
15   public List<Statement> getStatements()
16     { return statements; }
17   public void setStatements(List<Statement> statements)
18     { this.statements = statements; }
19
20   public String translateToC() {
21     String sb = "#include <stdio.h>\n";
22     sb += "main() {\n";
23     for(Statement stmt : statements) {
24       sb += stmt.toC() + "\n";
25     }
26     sb += "}\n";
27     return sb;
28   }
29 }
```

Statement.java

```
1 package gotolanguage;
2 import es.urjc.escet.metacet.annotation.Syntax;
3
4 @Syntax
5 public abstract class Statement {
6   public abstract String toC();
7 }
```

Figure 3-a. The Java annotated classes defining the **goto** language.

In figure 4 an example of parser usage is shown. A parser is instantiated (line 9) and then the method **parseProg()** (line 10) is called to parse a **Prog** program. The **parseProg()** method returns a **Prog** object which is the root of the tree. The **translateToC()** method is then called on the object (line 12) and the resulting C code is printed .

3.3. MetaCET's architecture

MetaCET is a Java-based tool for language design. It takes as input a set of annotated Java classes and, as a result, generates a scanner (which performs the target program lexical analysis), and a parser (which performs the target program syntactical analysis).

Print.java

```
1 package gotolanguage;
2 import es.urjc.escet.metacet.annotation.*;
3
4 @Syntax( " 'print' expression " )
5 public class Print extends Statement {
6   private String expression;
7
8   @Token("CTE")
9   public String getExpression()
10    { return expression; }
11  public void setExpression(String expression)
12    { this.expression = expression; }
13
14  @Override
15  public String toC() {
16    return "printf(\"%d\", " + expression + ");";
17  }
18 }
```

Labeled.java

```
1 package gotolanguage;
2 import es.urjc.escet.metacet.annotation.*;
3
4 @Syntax( " label ':' statement ")
5 public class Labeled extends Statement {
6   private String label;
7   private Statement statement;
8
9   @Token("ID")
10  public String getLabel() { return label; }
11  public void setLabel(String label)
12    { this.label = label; }
13
14  public Statement getStatement() { return statement; }
15  public void setStatement(Statement statement)
16    { this.statement = statement; }
17
18  @Override
19  public String toC() {
20    return label + ": " + statement.toC();
21  }
22 }
```

Goto.java

```
1 package gotolanguage;
2 import es.urjc.escet.metacet.annotation.*;
3
4 @Syntax( " 'goto' identifier " )
5 public class Goto extends Statement {
6   private String identifier;
7
8   @Token("ID")
9   public String getIdentifier() { return identifier; }
10  public void setIdentifier(String identifier)
11    { this.identifier = identifier; }
12
13  @Override
14  public String toC() {
15    return "goto " + identifier;
16  }
17 }
```

Figure 3-b. The Java annotated classes defining the goto language

```
1 package gotolanguage;
2
3 import impl.GotoLanguageParser;
4 import java.io.FileReader;
5
6 public class Main {
7
8   public static void main(String[] args) throws Exception {
9         GotoLanguageParser parser = new GotoLanguageParser();
10        Prog prog =
11          parser.parseProg(new FileReader("example.goto"));
12
13        String progInC = prog.translateToC();
14
15    System.out.println(progInC);
16  }
17 }
```

Figure 4. An example of parser usage.

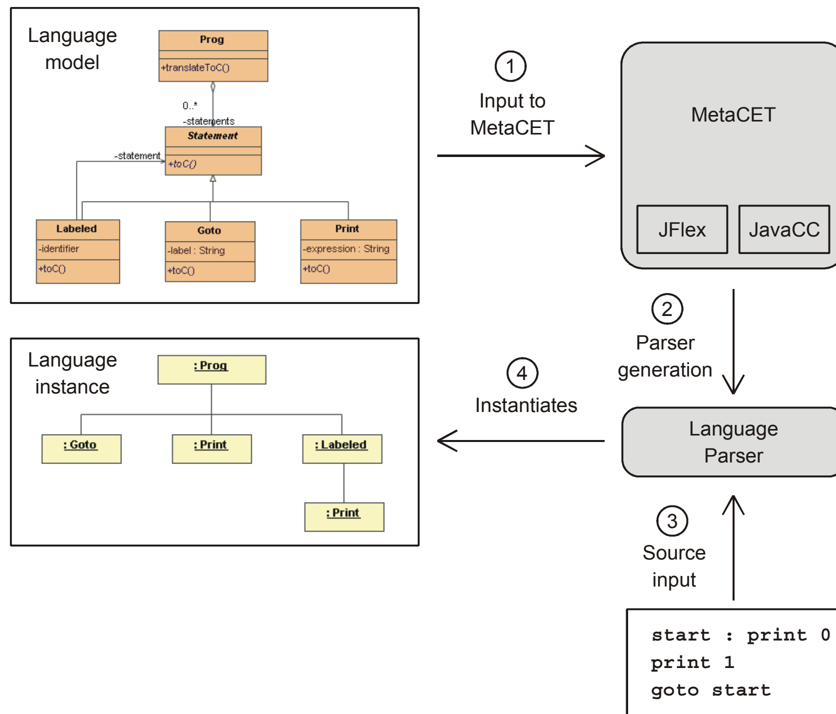

Figure 5. Architecture of MetaCET

The whole architecture of MetaCET is shown in Figure 5. As it can be seen in this figure, MetaCET generates a JFlex specification file for defining the scanner. Then, JFlex takes this specification as input for building the scanner.

A JavaCC specification file is also generated by MetaCET for defining the parser. Then JavaCC takes this specification as input for building the parser.

It is important to notice that MetaCET is not restricted exclusively to JFlex and JavaCC and it has been designed to allow the specification generation for other scanner and parser technologies.

The parser takes as input a target program and builds the tree by creating objects of the classes. Then, this tree of objects can be used for whatever purpose, such refactoring or translation. MetaCET generates the parser as a class. Its name is formed by the name of the language, followed by "Parser".

4. Discussion

In this section we explain the key advantages of defining a language with annotated Java classes against the use of specific languages.

- Taking into account features described in Section 2, in MetaCET the target language is modeled with Java classes. Regarding to syntax specification, in MetaCET is carried out by Java annotations. Finally, semantics are specified with Java code. Compared with the previous approaches, MetaCET doesn't need to learn any specific language for the task of language design.
- Use of annotations. Annotations, or metadata, allow programmers to decorate their code. These decorations can serve for multiple purposes. Annotations were introduced in the third edition of the Java language. Various technologies and tools have used annotations to specify aspects of classes and interfaces. Two Java technologies, that make intensively use of annotations, are EJB 3.0 (JSR 220), and JAXB 2.0 (JSR 222). Both of them use annotations to specify how Java objects are mapped to other persistent formats (databases and XML files, respectively). In MetaCET annotations are used the same way. That is, to specify how target programs (the persistent format) are mapped into Java objects.
- Easy to learn. Java programmers already know the syntax of classes. It is easy to learn MetaCET syntax in order to design languages. There is no need to learn new definition languages for concepts already known in Java.
- Total integration with any Java IDE. IDEs are very important tools in software development. They have features that make software development easy

in several ways. A language design in MetaCET is Java code, therefore the following features can be used without additional tools: syntax highlighting, real-time compilation, code completion or code assistance, quick error fix, navigation to declaration or code hyperlinking, search references or find usages, integrated standard documentation, code formatting, refactoring [12], among others. These features are available in IDEs for the most popular languages, but it is difficult to obtain them for language design. There is a lot of work in this area, but nowadays there is not a wide support.

- Easy to use. The way of access to the information of the parsed program is through *getter* and *setter* methods, which are standard in the object oriented paradigm.
- Language design focuses on relevant information. It is easy to recognize relevant information from the language model with MetaCET. For example, if the programmer defines a list of elements in the language that are comma-separated, the elements of the list are the relevant information. The commas are thrown out.

5. Conclusions and future work

This paper presents a new approach for language design using annotated Java classes as specification language. The presented tool generates a scanner and a parser that build a tree based on contents of the target program. Nodes of the tree are objects of the model classes.

This approach is more useful than other traditional language specification tools because of three main aspects. First, it is easy to design a new language because a well-known language is used. Also, the resulting tree is managed by the well-known JavaBeans getter methods semantics. Second, the language designer can be benefited from available Java IDEs and related tools. Such powerful environments are not commonly available for language design. Third, languages are defined in an object oriented way, which allows using associations and inheritance relationships. The difference to other tools is that the language definition is tightly integrated with the code that manages the tree.

The power of this language definition approach has been proved by means of designing a language model for the Java Language Specification, Third Edition. MetaCET has then been used to generate a parser for that version of the Java language.

There are more language related facilities that can be useful to manage the tree. These facilities can be incorporated in MetaCET. For example, visitor and traversal code, pretty-printers, and so on. Also, very useful tools could be generated automatically, like syntax highlighted editors for the target languages. New kinds of annotations could be defined for these purposes.

References

[1] Knuth, D. E.: History of writing compilers. Proceedings of the 1962 ACM national conference on Digest of technical papers, 43, ACM Press, 1962

[2] Wang, D. C., Appel, A. W., Korn, J. L., Serra, C. S.: The Zephyr abstract syntax description language. In Proceedings of the USENIX Conference on Domain-Specific Languages, 213—228, October 1997

[3] Visser, E.: Syntax Definition for Language Prototyping. PhD. University of Amsterdam, 1997

[4] Nystrom, N., Clarkson, M. R., Myers, A. C.: Polyglot: An Extensible Compiler Framework for Java. In Proc. 12th International Conference on Compiler Construction, Warsaw, Poland, April 2003. Lecture Notes in Computer Science 2622, 138—152.

[5] Garavel, H., Lang, F., Mateescu, R.: Compiler Construction Using LOTOS NT. In Proceedings of the 11th International Conference on Compiler Construction, London, UK, 9—13, Springer-Verlag, 2002.

[6] Anlauff, M., Kutter, P. W., Pierantonio, A.: Montages/Gem-Mex: A Meta Visual Programming Generator. In Proceedings of the IEEE Symposium on Visual Languages, Washington, DC, USA, 304, IEEE Computer Society, 1998.

[7] Kutter, P. W.: Montages, Engineering of Computer Languages. Ph.D. thesis, Swiss Federal Institute of Technology, 2004.

[8] Herndom, R. M., Verzins, V. A.: The realizable benefits of a language prototyping language. 803—809, 1988

[9] Spinellis, D., Guruprasad, V.: Lightweight Languages as Software Engineering Tools. {USENIX} Conference on Domain-Specific Languages, USENIX Association, Berkeley, CA, USA, 67—76, 1997

[10] Hedin, G., Magnusson, E.: The JastAdd system - an aspect-oriented compiler construction system, Science of Computer Programming 47, 37—58, Elsevier, 2003.

[11] Frankel, D. S.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Pub., Inc., 2003

[12] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999