Part I : Introduction to Polyglot with SwapJ

Raoul-Gabriel Urma - *Imperial College London*

This is a tutorial written for researchers and students getting started with using Polyglot to modify or extend Java. Many thanks to Prof. Sophia Drossopoulou, Prof. Nathaniel Nystrom and Prof. Andrew Myers for encouraging and reviewing this tutorial.

1 Polyglot

1.1 Introduction

Polyglot is a highly extensible compiler construction framework developed by Nystrom, Clarkson and Myers at Cornell University [1]. It performs parsing and semantic checking on a language extension and the compiler outputs Java source code. It is implemented as a Java class framework using design patterns to promote extensibility.

Several projects have successfully used Polyglot to extend the Java programming language; they range from large-to middle-scale projects. For example, *Jif* [2] is a language modification that extends Java with support for information flow control and access control, *SessionJ* introduces session-based Distributed Programming in Java [3] and J_0 is a subset of Java used for education [4].

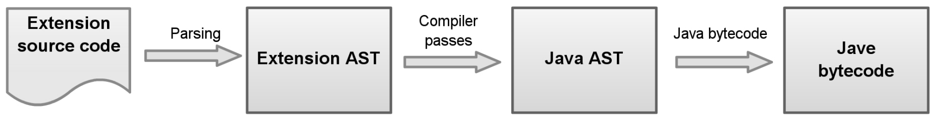
Language modifications follow the same pattern: they are implemented by extending the base grammar, type system and defining new code transformations on top of the base Polyglot framework. The result is a compiler that outputs Java source code. We can then compile the output with the standard Java compiler javac to generate bytecode runnable by the JVM.

Currently, Polyglot only supports Java version 1.4, but a language extension supporting most Java version 5.0 features has been developed [5].

Polyglot comes with the Polyglot Parser Generator *(PPG)*, a customised Look-Ahead LR Parser based on *CUP* [6, 7]. It is specially developed for the language extension developer to easily extend the Java base grammar defined with *CUP* by specifying the required set of changes [1]. In fact, *PPG* provides grammar inheritance, which enables the language extension developer to augment, modify or delete symbols and production rules from the Java base grammar.

The architecture of Polyglot follows standard compiler construction techniques. First of all, it first uses *JFlex*, a lexical analyzer generator [8], to perform lexical analysis on the source language. This step transforms the sequence of characters from the source code file into tokens. This chain of tokens is then parsed by *PPG*, which creates an abstract syntax tree (AST). An AST is a tree data structure that reflects the syntactic structure of a program. During the Polyglot process, this data structure is visited by several passes; the default set of passes include for example type checking, ambiguities removing and translation. In addition, Polyglot enables the introduction of extra passes in order to perform operations related to the compiler purposes: for example, optimising the AST. Finally, if no exceptions are thrown during the Polyglot process, the created compiler is expected to produce valid Java source code that can be compiled into Java bytecode.

Figure 1: Polyglot high-level process



1.2 In Details

In practice, implementing new language modification requires some knowledge about the Polyglot structure and internals. In this section, we explore Polyglot at a deeper level.

The latest revision of Polyglot can be fetched from the project SVN [9]. The Polyglot distribution contains several directories:

- **/bin/** : contains Polyglot base compiler and script *newext.sh* that generates the skeleton for a new language extension
- **/doc/** : various documentation about Polyglot
- **/examples/** : source codes of sample language extensions using Polyglot
- **/skel/** : skeleton directory hierarchy and files used for building new language extensions
- **/src/** : complete source code of Polyglot framework
- **/tools/java_cup/** : source code of tweaked version of Java CUP
- **/tools/ppg/** : source code of PPG

To create a language extension called [*extname*], the first step is to run **/bin/nexext.sh**, which creates the necessary skeleton package hierarchy and files:

- **[extname]/bin/[extname]c** : compiler for [extname]
- **[extname]/compiler/src/[extname]/** : source code for language extension
  - **ast** : AST nodes specific to [extname]
  - **extension** : Extension and delegate objects specific to [extname]
  - **types** : type objects and typing logic specific to [extname]
  - **visit** : visitors specific to [extname]
  - **parse** : symbols table, lexer and parser for [extname], *PPG* grammar file ([extname].ppg), lexer grammar file([extname].flex) The newext.sh script takes several parameter:

    Listing 1: newext.sh Parameters

    ```
    Usage: ./newext.sh dir package LanguageName ext
      where dir          - name to use for the top-level
          directory
                           and for the compiler script
            package      - name to use for the Java
                package
            LanguageName - full name of the language
            ext          - file extension for source files
    ```

  In addition, a class *ExtensionInfo.java* defines how the language extension will be parsed and type-checked. A file *Version.java* specifies the version of the language extension. Finally, the class *Main.java* brings all the parts together and performs the compilation.
- **[extname]/tests/** : sample [extname] source code test files

The second step is to define the language modifications. This is done by modifying the [extname].ppg file, which is processed by *PPG*. It specifies the changes to the base Java grammar required to generate a parser for the new language extension. Sometime the developer has to updated the lexer grammar file [extname].flex too, if new tokens are added. The full list of available instructions for the *PPG* grammar can be found on the *PPG* Project page [6]. They include:

- **extend S ::= productions**
  the specified productions are added to the nonterminal S.
- **override S ::= productions**
  the specified productions completely replace S

The *PPG* file also specifies how the parsed information is used to create a new AST. New AST nodes are instantiated through the language extension's NodeFactory, which has factory methods to create each AST node. This NodeFactory typically extends Polyglot's Java node factory, which is defined by the class `NodeFactory_c` class and implements the `NodeFactory` interface. This interface specifies all the factory methods for each node. Figure 2 depicts a UML diagram explaining the different classes involved in the node factory.

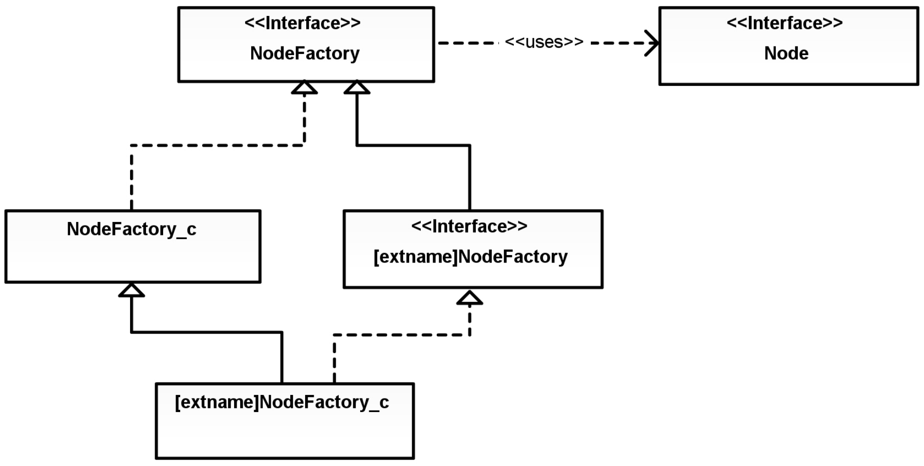


Figure 2: Language extension NodeFactory UML diagram
A language extension's node factory mechanism is split into two parts: an interface implementing the base node factory and a concrete class node factory extending the base concrete node factory.

The node factory can be accessed within the *PPG* file through the *parser.nf* instance. Listing 2 shows as an example the parsing and creation of an Assert Java base node.

Listing 2: Parsing and Creation of Assert AST Node

```
assert_statement ::=
ASSERT:x expression:a COLON expression:b SEMICOLON:d
{:
  RESULT = parser.nf.Assert(parser.pos(x, d), a, b);
:};
```

This snippet defines the production rule assert_statement. It is defined by an assert symbol and an expression representing the assert condition, a colon and another expression representing the error message followed by a semicolon. For example: assert(size == 0) is a valid assertion. If the parsing is successful, a new Assert AST node is created through the parser.nf.Assert(Position,Expr) node factory method.

After defining the syntactic changes through the grammar and defining the new AST node classes for the language modification, the next step is to specify the new semantic changes. New passes can be defined by defining and including them in *Extensions.java*. Most of the time, semantic changes can be implemented directly as part of the type-checking pass. Note that each node has a method `visitChildren(NodeVisitor v)` that is called before the type-checking process in order to disambiguate the types of the Node's fields. New nodes defined for the language extension must therefore also execute the visit on each fields. This is done by overriding `visitChildren (NodeVisitor v)` and passing the instance of the visitor to each field using the method `visitChild(Node,Visitor)`. For the sake of completeness, Listing 3 illustrates this mechanism and shows the method `visitChildren` of the Assert node.

Listing 3: Assert node's visitChildren(NodeVisitor) method

```
/** Visit the children of the statement. */
public Node visitChildren(NodeVisitor v) {
  Expr cond = (Expr) visitChild(this.cond, v);
  Expr errorMessage = (Expr) visitChild(this.errorMessage, v);
  return reconstruct(cond, errorMessage);
}
```

The Swap node has two fields: the condition expression (this.cond) and the error message (this.errorMessage). Both are passed to the visitChild method. The node is then reconstructed using the returned instances.

Type checking is done in each node by the method `typeCheck(ContextVisitor tc)` of a Node. If a type error exists the method throws a `SemanticException`. To continue with our example of the Assert node, Listing 4 illustrates type checking for an assert statement.

Listing 4: Assert node's type checking

```
public Node typeCheck(ContextVisitor tc) throws SemanticException {
        if (! cond.type().isBoolean()) {
            throw new SemanticException("Condition of assert statement " +
                                        "must have boolean type.",
                                        cond.position());
        }

        if (errorMessage != null && errorMessage.type().isVoid()) {
            throw new SemanticException("Error message in assert statement " +
                                        "cannot be void.",
                                        errorMessage.position());
        }

        return this;
    }
```

The method type-checks if the expression cond is of type boolean qnds if the expression *errorMessage* is defined and not void.

In addition, the language developer can access the `TypeSystem` instance through the `ContextVisitor`. The `TypeSystem` instance defines the types of the language and how they are related. For example, it is used to compare the equality between two types, set new types on the expressions or check if a type can be cast to another type. Listing 5 shows an example of using the `TypeSystem` from the `ContextVistor`.

Listing 5: Switch_c's node typechecking

```
 /** Type check the statement. */
public Node typeCheck(ContextVisitor tc) throws
    SemanticException {
  TypeSystem ts = tc.typeSystem();
  Context context = tc.context();

  if  (!ts.isImplicitCastValid(expr.type(), ts.Int(), context)
      && !ts.isImplicitCastValid(expr.type(), ts.Char(),
          context))
    {
      throw new SemanticException("Switch index must be an
          integer.", position());
    }
        return this;
}
```

The index of a switch statement (switch(index)) can only be a char type or an integer type. However, any type that can be cast to an int or a char is also allowed. For example, an Integer or a short is valid but a String isn't. The Switch_c's typeCheck method gets the type system from the instance tc of the ContextVisitor and then calls the method isImplicitCastValid(Type, Type, Context) to perform the casting checks.

The final step after the semantic analysis of each AST node is to produce valid Java code. There are several options available to the language extension developer.

First, the most extensible way is to introduce a separate pass that transforms the language extensions AST nodes to valid Java AST nodes and then rely on the default Java AST translation pass to output valid Java source code. New passes can be added by extending the default Polyglot scheduler: `JLScheduler`. One would then have to create a pass by extending an appropriate Polyglot visitor class and schedule the pass before the `CodeGenerated` pass, which is responsible for translation. The created pass will be responsible for rewriting language extensions AST nodes into Java AST nodes using the `NodeFactory` methods. In addition, one can also use the Polyglot *quasiquoting* feature, which enables the generation of Java AST nodes from a String (*polyglot.qq.QQ*). This standard technique ensures that the output is well-formed Java code.

Secondly, another way to translate to Java code is to simply override the method `prettyPrint(CodeWriter, PrettyPrinter)` of each new Node. This method is responsible for printing the generated code to the output file and is called by the default implementation of the method `translate(CodeWriter, Translator)`, which is called during the Translation pass. Although this is a quick and easy way to perform code generation, it makes it harder to extend the modified language later. Furthermore, Polyglot won't check that the generated Java code is valid, so errors may show up when the code is compiled.

As an example, Listing 6 shows the code generation for the Assert AST node and Listing 7 shows the code generation for the Throw AST node.

Listing 6: Assert node translation

```
 /** Write the statement to an output file. */
public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
  w.write("assert ");
  print(cond, w, tr);

  if (errorMessage != null) {
    w.write(": ");
    print(errorMessage, w, tr);
  }

  w.write(";");
}

public void translate(CodeWriter w, Translator tr) {
  if (! Globals.Options().assertions) {
    w.write(";");
  }
  else {
    prettyPrint(w, tr);
  }
}
```

The translate method from the Assert node by default calls the prettyPrint method which handles the code generation to the output file. Note that the print(Node child, CodeWriter w, PrettyPrinter pp) method will handle the code generation for the Node instance child. Essentially it calls its prettyPrinter method.

Finally, the language extension compiler is ready and can be used by running *Main.java*.

Listing 7: Throw node translation

```
/** Write the statement to an output file. */
public void prettyPrint(CodeWriter w, PrettyPrinter tr) {
  w.write("throw ");
  print(expr, w, tr);
  w.write(";");
}
```

Similarly to the Assert node, the Throw node's prettyPrint method handles code generation and writes code to the output file.

2 SwapJ

In this section, we bring the concepts introduced about Polyglot together to show how to create a compiler for a language extension. We create *SwapJ*, a language that extends Java with a swap functionality. The modification made to Java is simple: we introduce a new swap(x,y) keyword that swaps the contents of the arguments x and y if they are of the same type. We don't support swapping array accesses and field accesses, for simplicity.

We start by formally defining the syntax changes to the Java base grammar and also provide the semantics and type systems for the swap operation. After, we work step by step and implement the *SwapJ* compiler.

2.1 Formal Definition

We describe the syntax of our new build-in swap operation:

Listing 8: SwapJ BNF

```
<statement> ::= "swap" "(" <identifier> "," <identifier> ")"
    ";"
                | <statement>
```

We define the operational semantics of our swap operation. Swapping the two variables x and y means to assign the content of y to x and assigning the content of x to y in the store ϕ:

Swap$_{OS}$

$$\frac{\mathsf{stmts}, \phi[\mathsf{x} \mapsto \phi(\mathsf{y}), \mathsf{y} \mapsto \phi(\mathsf{x})], \chi \rightsquigarrow \mathsf{v}', \chi'}{\mathsf{swap}(\mathsf{x}, \mathsf{y}); \mathsf{stmts}, \phi, \chi \rightsquigarrow \mathsf{v}', \chi'}$$

We also define the type rule of our swap operation:

Swap$_{TS}$

$$\frac{\begin{array}{l}\mathsf{P}, \Gamma \vdash \mathsf{x} : \mathsf{t} \\ \mathsf{P}, \Gamma \vdash \mathsf{y} : \mathsf{t}\end{array}}{\mathsf{P}, \Gamma \vdash \mathsf{swap}(\mathsf{x}, \mathsf{y}) : \mathsf{void}}$$

2.2 Implementation

1. **Build skeleton extension**
   As explained in the previous section, the first step is to run *newext.sh* to build the skeleton files and directories for our language extension:

   Listing 9: Creation of SwapJ files structure

   ```
   ./newext.sh swapJ swapj SwapJ sj
   ```

   The directory containing the skeleton file structure is SwapJ. The package name is swapj, the language name is SwapJ and the extension file for our SwapJ source files is .sj

2. **PPG grammar specification**
   The next step is to specify the syntactic grammar differences to the Java base grammar. We translate our BNF specification into *PPG* grammar and specify the changes in swapJ.ppg (Listing 10) as explained in the previous section. Since we are adding a new token `swap` we also have to modify the lexer grammar file (Listing 11).

   Listing 10: PPG grammar for SwapJ

   ```

   terminal Token SWAP;
   non terminal Stmt swap_stmt;

   start with goal;

   swap_stmt ::= SWAP:a LPAREN name:l COMMA name:r RPAREN
       SEMICOLON:b {:
     RESULT = parser.nf.Swap(parser.pos(a,b),l.toExpr(), r.
         toExpr());
   :};

   extend statement_without_trailing_substatement ::=
       swap_stmt:a {: RESULT = a; :};
   ```

   We extend the Java statement and add a new Swap statement. Note that we also added a token SWAP that is defined in the lexer grammar file.

   Listing 11: Lexer grammar for SWAP token

   ```
    keywords.put("swap",          new Integer(sym.SWAP));
   ```

3. **NodeFactory and AST Nodes**

   The next step is to define the new SwapJNodeFactory and create the necessary AST node. Listing 10 shows that if parsing is successful, a node `Swap` will be created through the node factory. We now need to specify the interfaces required, implement the concrete classes and follow UML diagram 2 describing the AST node creations.

We create an interface `Swap` (Listing 12), a concrete class `Swap_c` (Listing 13) providing the implementation, a new factory interface `SwapJNodeFactory` (Listing 14) which provides the template for a swap node creation but also implements the base `NodeFactory` interface and the concrete node factory `SwapJNodeFactory_c` (Listing 15) that handles the instantiation of concrete Swap nodes. UML diagram 3 summarises the hierarchy and relations between the different classes.

Listing 12: Swap interface

```
public interface Swap extends Stmt{

}
```

A swap node is a statement and therefore implements the Stmt interface.

Listing 13: Swap_c concrete class constructor

```
public class Swap_c extends Stmt_c implements Swap{

  private Expr left_e;
  private Expr right_e;

  public Swap_c(Position pos, Expr left_e, Expr right_e) {
    super(pos);
    this.left_e = left_e;
    this.right_e = right_e;

  }
```

The constructor of a Swap node takes the Position in the source file from the parser and also two expressions representing the left variable and right variable to swap. The Swap_c also extends the Stmt_c class, which encapsulates behaviour of any Java statement.

Listing 14: SwapJNodeFactory interface

```
/**
 * NodeFactory for swapJ extension.
 */
public interface SwapJNodeFactory extends NodeFactory {

  Swap Swap(Position pos, Expr left_e, Expr right_e);

}
```

Listing 15: SwapJNodeFactory_c concrete class

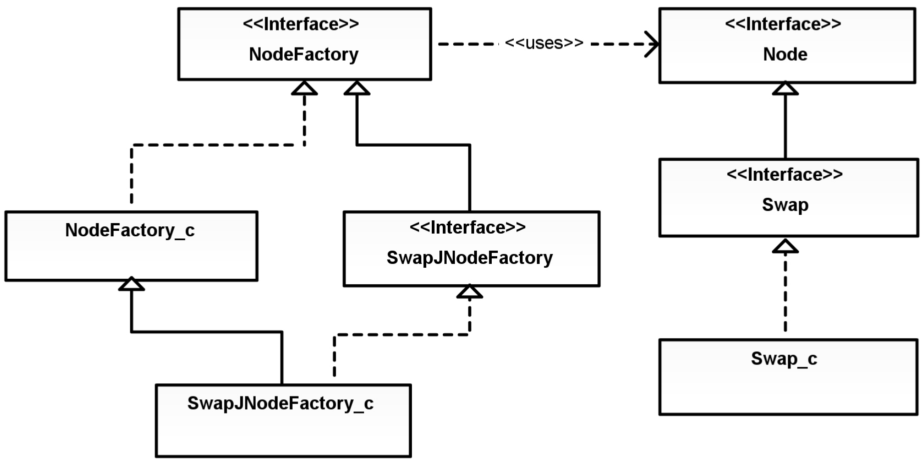
```java
/**
 * NodeFactory for swapJ extension.
 */
public class SwapJNodeFactory_c extends NodeFactory_c
    implements SwapJNodeFactory {

  public Swap Swap(Position pos, Expr left_e, Expr right_e
      ) {
    return new Swap_c(pos, left_e, right_e);
  }

}
```

The factory method Swap(Position, Expr, Expr) returns a concrete node Swap_c

Figure 3: SwapJ class diagram



4. **Semantic changes**
The next step is to perform type checking on the swap operation: we need to ensure that the two arguments to swap are of the same type. As explained in the previous section, type checking is performed by the `typeCheck(ContextVisitor)` method of each node. We override this method so it checks if the two expressions of the Swap node are of the same type. Note that we also need to override the method `visitChildren(NodeVisitor)` to ensure the types of the Swap node arguments are disambiguated. Listing 16 shows how the `Swap_c` concrete class' methods `visitChildren` and `typeCheck` are implemented.

Listing 16: Swap node type checking

```java
public Swap reconstruct(Expr expr_l, Expr expr_r) {
  if (this.left_e != expr_l || this.right_e != expr_r) {
    Swap_c n = (Swap_c) copy();
    n.left_e = expr_l;
    n.right_e = expr_r;
    return n;
  }
  return this;
}

@Override
public Node visitChildren(NodeVisitor v) {
  Expr expr_l = (Expr) visitChild(left_e, v);
  Expr expr_r = (Expr) visitChild(right_e, v);

  return reconstruct(expr_l, expr_r);
}

@Override
public Node typeCheck(ContextVisitor tc) throws
    SemanticException {

  SwapJTypeSystem ts = (SwapJTypeSystem)tc.typeSystem();

  Type left_t = left_e.type();
  Type right_t = right_e.type();

  if(!left_t.typeEquals(right_t))
  {
    throw new SemanticException("swap() arguments of
        different types!");
  }

  return this;

}
```

The overriden visitChildren disambiguate the Swap_c's fields and return a disambiguated Swap node. The overriden typeCheck method uses the type systems to verify the type equality between the two swap's arguments.

5. **Translation and code generation**
   The final step is translation. Since the swap translation to Java source code is straightforward, we can override the `prettyPrint(CodeWriter w, PrettyPrinter tr)` method directly and define code generation as explained in the previous section. We also need a fresh variable name to perform the swap, and Polyglot provides a helper static method for this: `Name.makeFresh()`. Listing 17 shows how to perform the code generation of a Swap node.

Listing 17: Swap_c node's code generation

```
@Override
public void prettyPrint(CodeWriter w, PrettyPrinter tr) {

  // fresh variable name
  String fresh = Name.makeFresh().toString();

  // int temp = y;

  left_e.type().print(w);
  w.write(" " + fresh + " = ");
  print(right_e,w,tr);
  w.write(";\n");

  // y = x;
  print(right_e,w,tr);
  w.write(" = ");
  print(left_e,w,tr);
  w.write(";\n");

  // x = y;
  print(left_e,w,tr);
  w.write(" = " + fresh);
  w.write(";\n");
}
```

6. **Testing**
   The *SwapJ* compiler is now operational, and we can test code generation. We create a swapTest class written in *SwapJ* and compile it with the *SwapJ* compiler. Listing 18 shows the class written in SwapJ and Listing 19 shows the generated Java output.

Listing 18: swapTest class written in SwapJ

```
public class swapTest {

    public static void main(String[] args) {

  String x = "Swapj!";
  String y = "Java";

    swap(x,y);
    // Java Swapj!
    System.out.println(x + " " +y);
    swap(x,y);
    // Swapj! Java
    System.out.println(x + " " +y);
    }
}
```

Listing 19: swapTest class after compilation

```
public class swapTest {

  public static void main(String[] args) {
    String x = "Swapj!";
    String y = "Java";
    java.lang.String id0 = y;
    y = x;
    x = id0;

    System.out.println(x + " " + y);
    java.lang.String id1 = y;
    y = x;
    x = id1;

    System.out.println(x + " " + y);
  }
}
```

2.3 Summary

We showed how to quickly implement a simple language extension by using Polyglot. This section was written as a tutorial for researchers and students interested to develop language extensions. In the next part of the tutorial, we will go in further depth and explore the scheduling of passes in Polyglot: we will add an AST Rewriting pass to directly transform SwapJ nodes into Java nodes. Table 2.3 summaries the lines of code added for each new class introduced to implement the *SwapJ* language extension.

Figure 4: SwapJ code modifications summary

Classes	Lines of code
SwapJNodeFactory	14
SwapJNodeFactory_c	14
Swap	7
Swap_c	117
Total	152

References

[1] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. Technical report, Cornell University, 2003.

[2] Nate Nystrom, Lantian Zheng, Steve Zdancewic, Andrew Myers, Stephen Chong, and K. Vikram. Java information flow. `http://www.cs.cornell.edu/jif/`.

[3] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. Technical report, Imperial College London, 2008.

[4] Andrew Jonas, Daniel Lee, and Andrew Myers. J0: A Java extension for beginning (and advanced) programmers. `http://www.cs.cornell.edu/Projects/j0/`.

[5] Milan Stanojevic and Todd Millstein. Polyglot for Java 5. `http://www.cs.ucla.edu/~todd/research/polyglot5.html`.

[6] Michael Brukman and Andrew C. Myers. A parser generator for extensible grammars. `http://www.cs.cornell.edu/projects/polyglot/ppg.html`.

[7] Princeton University and Technical University of Munich. LALR parser generator for java. `http://www2.cs.tum.edu/projects/cup/`.

[8] JFlex - the fast scanner generator for Java. `http://jflex.de/`.

[9] Polyglot svn. `http://polyglot-compiler.googlecode.com/svn/trunk/polyglot/`.

List of Figures

Listings