HiLexed – parsing in context

Reference guide

Written by Anders Björklund

Document version 1.0

Chapter 1

Introduction to HiLexed

HiLexed is a left-to-right, leftmost derivation (LL), unlimited lookahead parser system with on-demand compilation and context sensitive lexing. HiLexed is not a parser compiler, but a dynamic component system that parsers are made of. This way, parsers can be used to create higher order parsers. Serialized HiLexed parsers exhibit linear performance O(n).

Left-to-right, leftmost derivation

LL-parsers, function in the way of having their initial context set to the root of the document, that is, starting with the general view of a document and going further and further into subsections.

Unlimited lookahead and unlimited lookbeyond

By having unlimited lookahead, HiLexed ensures that it will not need to use backtracking when deliberating what to parse next. The lookahead components, structure themselves into a network using references. This way, the system can look beyond the end of a rule without growing exponentially as it only has to consider specific situations once.

On-demand compilation

HiLexed will compile semantic actions specified by the developer, but only if the code has changed since last invocation. There is no separate compilation step.

Context sensitive lexing

Lexing is the process of grouping and classifying sets of characters in a document. By having lexing as an integrated part of the parse, the parser can delay the classification to a time when context information is available. The parser knows at any time what terminals are allowed next, since it knows exactly where in the document structure it is.

Higher-order parsers

The basic parser component of the system is called a segment. A segment can behave as either a terminal or a non-terminal in parsing terms. Segments can dynamically be connected to each other and their behavior made to change. Because of this, a parser (with semantic actions) can build a parser.

The EBNF-parser is an example of such a parser. It was created using the base API with the explicit intent of removing the need to call the base API directly. It makes possible the use of a different, and much more appropriate language for creating parsers.

Other such parsers are possible to create using the HiLexed base API, but they can be created much faster and can be maintained much easier using the EBNF parser.

Even if you fall flat on your face·at least you are moving forward.
Unknown

Chapter 2

Up and running

HiLexed is written in java and executes on the Java Virtual Machine, JVM. It needs to have write access to a cache directory and there are certain classes that need to be included on the classpath.

The simplest way to test HiLexed is using an IDE such as Eclipse or NetBeans.

Java virtual machine

HiLexed needs version 1.6 (or better) of the Java Development Kit, JDK.

The cache folder

The cache folder is where HiLexed stores dynamically compiled components such as semantic actions and serialized parsers.

The default location is <your project root>/cache.

Classpath settings

Since HiLexed compiles certain blocks of code dynamically, a file called “tools.jar” needs to be included on the classpath. This file is usually found in <java installation folder>/lib.

The cache folder also needs to be included on the classpath as your parser needs to be able to find, load and execute semantic actions.

Eclipse users

(2 minutes)

1. Create a java project called “HelloSyntax”, using the Eclipse guide. Click “next” once.
2. Select the “Libraries” tab.
3. Click "Add external jar" and select the file “HiLexed-2.x.y.jar”
4. Click "Add external jar" and select the file “tools.jar”, which you will find in the “lib” subfolder of your JDK-install folder.
5. Click "Add class folder". Select the “HelloSyntax” project and create a new folder called “cache”. This folder should be located in the project root.
6. Finish the project.

Everything should now be ready for you to proceed to the next chapter – HelloSyntax.

There is nothing better for understanding a kangaroo than seeing it jump.
Dick Grune and Ceriel J.H. Jacobs, “Parsing Techniques”

Chapter 3

Hello syntax

The recommended way of using HiLexed is to specify your language or document structure using a syntax known as Extended Backus-Naur Form, EBNF. By feeding your syntax file into the included EBNF-parser, this will return your parser, normally without any more work needed on your part. The specification becomes your parser.

NOTE: If you try the example below, please note that a simple cut and paste may insert control characters from the PDF itself.

First contact

Create a new java project (see previous chapter).

Create a package called hilexed.test.

Create a syntax definition file (a regular text file) in this package and name it hello.syntax.

hello.syntax

```
greeting : (“Hello” | “Goodbye”)+ “world” ;
```

Create a source file, a.k.a. the document you would like to parse, called hello.source and save it in the same package.

hello.source

```
Hello world
```

Next, create a new java class called HelloSyntax in the same package.

HelloSyntax.java

```java
package hilexed.test;

import org.hilexed2.base.IExpression;
import org.hilexed2.base.Parser;
import org.hilexed2.parsers.Ebnf;
import org.hilexed2.utilities.Source;

public class HelloSyntax {
  public static void main(String[] args) {
    IExpression ebnf = new Ebnf();
    ebnf.put(Parser.SYNTAX, Source.readJar("/hilexed/test/hello.syntax"));
    IExpression myParser = (IExpression) ebnf.invoke(null);

    myParser.put(Parser.SOURCE, Source.readJar("/hilexed/test/hello.source"));
    myParser.put(Parser.ECHO, Boolean.TRUE);
    myParser.invoke(null);
  }
}
```

Now compile and run HelloWorld.

If everything was set up correctly you should see the following output.

```
[TERMINAL]: Hello
[TERMINAL]: world
```

Try changing the *hello.source* file perhaps by inserting '*Goodbye*' at various locations or by inserting “garbage” characters or words not described in the *hello.syntax* file.

In the example, the generated parser would expect one or more of '*Hello'* and '*Goodbye'* (in arbitrary order) and that this sentence would always end with '*world'*. If the input does not follow the syntax definition, the parser will respond by displaying the offending text fragment and then go on parsing as normal. If unable to find any matching text at all starting from its current position, it will report this as a lookahead error.

EBNF

Extended Backus-Naur Form is a compact syntax for describing syntax. Rules are defined using a name, references to other rules, groups and terminals, along with quantifiers which describe option, repetition and choice.

Names

A name is an unlimited-length sequence of letters, digits, currency symbols and underscores, the first of which may not be a digit. Names must not be java keywords.

References

Rules defined elsewhere in the syntax definition are referenced by simply including their name in a rule.

Groups

Groups are sequences of terminals and references contained inside a parenthesis. They can be arbitrarily deeply nested.

Terminals

Regular expressions contained inside quotation marks. Backslash \ functions as escape character. Using the escape character, you can include quotation marks and backslashes inside a terminal without inadvertently ending the terminal.

Segments

In situations where it is not important to distinguish between references, groups and terminals, these are all simply referred to as "segments".

Quantifiers

Quantifiers can be attached to any type of segment and determines how may times to match against it. The default quantifier for any type of segment is "once".

? optional segment.

***** zero or more segments

+ one or more segments

| choice of segments (either of)

Example

```
packageDecl : “package” (identifier “\.”)* identifier ;

identifier : “[a-zA-Z]+” ;
```

The following table describes the segments of the packageDecl rule.

Example	**Type**	**Use**
packageDecl	name	Used when referring to this rule from other rules
“package”	terminal	Regular expression
( )	group	Contained segments are treated as a single entity
identifier	rule reference	References the rule named “identifier”
“\.”	terminal	Regular expression. A single dot. (Escaped)
*	quantifier	Group can appear zero or more times
identifier	rule reference	Second reference to rule “identifier”
;	rule end	

Chapter 4

Hello semantics

Chapter 5

Growing a Tree

Chapter 6

Higher level constructs

Chapter 7

Low level constructs

When you have eliminated the impossible, whatever remains, however improbable, must be the truth. -- Sherlock Holmes, "The Sign of Four"

Chapter 8

Lookahead