The Clump language Reference Manual v0.3

Didier Plaindoux

October 5, 2010

Contents

Introduction

In Class-Based programming languages the separation between types and Classes increases the language expressiveness. But in such Object-Oriented approach the language provides an implicit way for objects specification and implementation through classes or dedicated syntactic construction in Trait based languages. The main consequence was the design dependency existing between the object internal state and behaviors defined in a class. Therefore modifying a state specification implies a behavior modification and vice versa.

In this document we propose a natural evolution of the Scandinavian Object-Oriented paradigm revisiting the duality data/knowledges. Thus a new approach where both Class-Oriented and Pattern-Oriented approaches in a same language is proposed.

License

Availability

The complete Clump distribution is hosted by SourceForge and can be accessed via the `http://vorpal.sourceforge.net` web site and sources can be accessed via the `http://vorpal.svn.sourceforge.net/viewvc/vorpal/` web site. Beware poject in SourceForge is hosted using the vorpal project name but while this name is a registered trademark of the University of Colorado it cannot be used for the diffusion and the language with is corresponding copyleft.

Finally this documentation only covers language definitions available in the version 0.3 of the language.

Chapter 1

The Clump syntax

The Clump language syntax is inspired by C, C++ and Java^{TM} languages for better comprehension based on the same syntax convention. From this starting point we propose now a step by step language description and specification.

1.1 Lexical conventions

1.1.1 Blanks

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed. These characters as no specific signification and are ignored. Nevertheless these definitions are mandatory while blanks are the only way to separate literals in the source code. In the following section all rules are defined with implicit definition of blanks just in order to propose a simple and comprehensive syntax.

1.1.2 Comments

Comments are introduced by two conventions inspired by the Java^{TM} and C languages. First one is a line based comment when the second is a block based comment. In the following sections all rules are defined with implicit definition of blanks just in order to propose a simple and comprehensive syntax.

```
comment   ::=  // (char - {\n})* \n?
               /* char* - {*/} */
```

There is no way to specify structural comment like Java andits corresponding mechanism called javadoc. The current version of the syntax does not provide any support for documentation construction for API definition and navigation.

Encapsulated block commment in another one is not allowed. In that case a comment like `/*1 ... /*2  ... 2*/ ...1*/` starts at token `/*1` and ends at

2*/. Therefore all characters following this comment are considered as a Clump source code fragment.

Examples

```
/* This is a simple
   multi-line comment */

// And this is a simple line comment
```

1.1.3 Integer literals

An Integer is a sequence of one or more digit. A negative number is specified with the minus sign at the beginning of the literal. Default supported integers are in decimal (radix 10). An alternative syntax is proposed for hexadecimal (radix 16) integer definition; these integers are preceded by the 0x prefix or 0X prefix.

```
int        ::=  (-)?(digit)+
                (-)?(0x|0X)(digit|a..f|A..F)+
digit      ::=  0..9
```

Examples

```
123, -9, 0xFF3E, 0XA3, -0xFE, -0Xab
```

1.1.4 Character literals

A character literal is a single character delimited by the single quote character. A character literal can be a regular character or an escaped one.

```
character  ::=  'char - {'}'
char       ::=  (regular|escaped)
escaped    ::=  \(b|r|n|t|f|'|")
```

The syntax prohibits the ''' sequence because it's interpreted as ('')'. Then the character literal does not enclose any character and the last quote has not corresponding ending quote which implies an unbalanced term. For this purpose escaped characters are defined then the right syntax for the previous character literal is ''. Indeed some characters can also be denoted usign this escape sequence as usual in a programming language.

Examples

```
'a', '\t', '\''
```

1.1.5 String literals

A string is a sequence of characters delimited by double quote character.

```
string    ::=  "(char - {"})*"
```

The syntax prohibits the `"""` sequence because it's interpreted as `("")"`. The empty string is valid but the last double quote don't imply a well balanced term and then it's not a well formed sequence of literals.

Examples

```
"a simple string", "Another \"one\"", "A multi \n line string"
```

1.1.6 Reserved keywords

The following set of identifiers are keywords in the language. These identifiers cannot be used as conventional identifiers in the language.

package	import	object	type	interface	trait	class
abstract	final	implements	extends	case	this	
thistype	self	selftype	new	with	return	do
while	for	if	unless	else	throw	throws
try	catch	finally	switch	default	as	

Four set of keywords are identified. The first one - corresponding to the first line - contains all keywords used for the entity declaration.

1.1.7 Identifiers and operators

Identifiers are used when specifying any kind of entity or variable in the language.

```
fullid    ::=  id (. id)*
id        ::=  (a..z|A..Z)(a..z|A..Z|0..9|_)*
operator   ∈   {&,|,&&,||,<,>,<=,=>,==,!=,+,-,*,/, %, :=}
```

1.1.8 Type specification

Type specification is built upon a name and a set of type variable. Each type variable can be bounded or not.

```
tspec     ::=  id tspecs?
tspecs    ::=  < tvar (, tvar)* >
tvar      ::=  id (extends tinst)?
```

Examples

```
Hash<E,V>
Cons<E extends Peano>
int
Comparable<E extends Comparable<E>>
```

1.1.9 Type instance

A type instance references defined type where all parametric types are mandatory. A special construction provides convenient syntax for arrays declaration but this is just a macro.

```
tinst      ::=  thisinst? fullid tinsts?  []*
                { tinst ( (tinst (, tinst)*)? )  exc?  }
thisinst   ::=  < tinst >
tinsts     ::=  < tinst (, tinst)* >
```

Examples

```
int
String
List<int>
Hash<int,List<String>>
<List<int>> Cons<int>
{int (int,int) throws DivideByZeroException}
List<int>[]
Array<List<int>>
```

1.2 Statements and Expressions

1.2.1 Statements

```
stmt       ::=  stmt stmt
                { stmt } (;)?
                expr ;
                type id = expr ;
                fullid = expr ;
                if ( expr ) stmt (else stmt)?
                unless ( expr ) stmt (else stmt)?
                while ( expr ) stmt
                do stmt while ( expr ) ;
                for ((tinst? id=expr)?; expr; (id=expr)?)  stmt
                for (tinst id:expr) stmt
                switch ( expr ) { cases }
                return expr ;
                throw expr ;
                try { stmt } catches
cases      ::=  case tinst (as id)? :  stmt cases?
                default :  stmt
catches    ::=  catch ( tinst id ) { stmt } catches?
                finally { stmt }
```

Examples

```
int i = 1;
this = null;
if (true) { int i = 2; }
if (true) { /* empty */ } else { int i = 3; }
while(i < 1000) { i = i + 1 }
do { i = i + 1 } while (i < 1000);
for(int i = 0; i < 1000; i = i + 1) { /* i++ does not exist */ }
for(int i : listofint) { ... }
throw new Exception();
try { ... } catch (Exception e) { ... } finally { ... }
switch (e) { case T1 as v: ... default: ... }
```

1.2.2 Expressions

```
expr       ::=  integer
                string
                char
                ( expr )
                this (with { id = expr (; id = expr)* })?
                self
                expr .  id
                expr .  call
                new (< tinst >)? fullid (< tinsts >)? ( parms )
                expr operator expr
call       ::=  fullid (< tinsts >)? ( (expr (, expr)*)? )
```

Examples

```
123
"Hello world !"
'\n'
new <Peano> Succ(new Succ(new Zero()))
this with { att = new <Peano> Succ(new Succ(new Zero())) }
self
self.m(this.att)
123 + x
```

1.3 Objects

```
object     ::=  private? modif? object tspec extends? { objectspec* } (;)?
modif      ::=  abstract
                final
objectspec ::=  attrib
                methspec { stmt {
                constr
extends    ::=  extends tinst (, tinst)*
attrib     ::=  private? final? tspec id (, id)* ;
constr     ::=  id ( tparams? )  super? { stmt }
tparams    ::=  tinst id (, tinst id)*
super      ::=  :  call (, call)*
```

Examples

```
object Zero { /* Empty object - Kingdom of nouns :-) */ }

object Succ {
    final Peano value;

    Succ(Peano value) {
        this.value = value;
    }
}

object Point {
    int x,y;
    Point() {
        this.x = 0;
        this.y = 0;
    }
}
```

1.4 Types

```
type       ::=  private? type tspec = tinst (| tinst)* (;)?
```

Examples

```
type Peano =
  Zero
| Succ
```

1.5 Interfaces

```
interface  ::=  private? final? interface tspec extends? { (methspec ;) * } (;)?
methspec   ::=  tinst tspec ( (tinst (, tinst)*)? )  exc?
                tinst ( operator ) ( (tinst (, tinst)*)? )  exc?
exc        ::=  throws tinst (, tinst)*
```

Examples

```
interface IPeano<T> {
    Peano add(Peano);
    String toString();
}
```

1.6 Classes

```
class      ::=  private? modif? class tspec( tinst ) cextends? implements? { case* } (
cextends   ::=  extends cinst (, cinst)*
cinst      ::=  tinst (as id)?
implements ::=  implements tinst (, tinst)*
case       ::=  case tinst methods
                method
methods    ::=  method
                { method method* }
method     ::=  methspec { stmt }
                tspec ( tparams? )  { stmt }
                ( operator ) ( tparams? )  { stmt }
mparams    ::=  tinst? id (, tinst? id)*
```

Examples

```
class coloredPoint(ColoredPoint)
        extends point as superPoint,
                color as superColor
        implements IPoint<thistype>, IColor
{
    // Protected method
    String name() {
        return "Colored"
    }
    toString() {
        return name() + "{" + superPoint.toString() + ", " + superColor.toString() + "}";
    }
}
```

1.7 Traits

```
trait      ::=  private? trait tspec implements? { method* } (;)?
```

Examples

```
trait comparable<E> implements Comparable<E> {
    (!=)(e) { return (self == e).not(); }
    // (==)(e) is not provided
}
```

1.8 Compilation unit

```
unit       ::=  package? import* entity*
package    ::=  package fullid (;)?
import     ::=  import fullid (;)?
entity     ::=  object
                interface
                type
                class
                trait
                type
```

Examples

```
package samples.peano

object Zero {}

class zero(Zero) implements IPeano {
    add(p) { return p; }
    toString() { return "0"; }
}
```

Chapter 2

The Clump semantic

In this chapter we focus on the language semantic. It covers language specific functionalities but also revisited concepts.

2.1 Extension vs. Variant approaches

In the language two mechanisms dedicated to the type definition are identified. The first one came from the Class-Centric approach and was embodied by the object extension mechanism. The second one is the capability to specify variant types inspired by Data-Centric programming languages.

2.1.1 Traditional Object-Oriented approach

The first approach is based on a well known mechanism in Object-Oriented languages which is the extension. In Clump an object can be extended easily with a similar mechanism as shown in the following example.

Examples

```
object abstract Arithmetic {}

object Integer extends Arithmetic {
    int value;
    Integer(int value) {
        this.value = value;
    }
}

object Plus extends Arithmetic {
    Arithmetic left, right;
    Plus(Arithmetic left, Arithmetic right) {
        this.left = left;
        this.right = right;
    }
}
```

Therefor arithmetic terms can be extended easily designing object which are `Arithmetic` extensions. This is the traditional Object-Oriented approach.

This extension defines a partial order over object type domains defining explicit subtyping property used during type checking stage. Indeed it means that when an `Arithmetic` object is required it de-facto covers all the existing and future extensions. Such subtyping approach is also called nominative subtyping and is influenced by the extension declarations only.

2.1.2 Variants type

Types can be designed independently and these type are grouped using variant types. Therefor each type cannot be compared with other type specified in the variant. But like extension based type when a method is specified with a variant it accepts type defined in this variant and nothing else. Back to the arithmetic term problematic the previous specification can also be proposed using variants.

Examples

```
object Integer {
    int value;
    Integer(int value) {
        this.value = value;
    }
}

object Plus {
    Arithmetic left, right;
    Plus(Arithmetic left, Arithmetic right) {
        this.left = left;
        this.right = right;
    }
}

type Arithmetic = Integer | Plus
```

The main difference is the ability to specify methods with abstract type and therefor open type and variant type which is by opposite definitive. The last case combined with a view approach based on cased method provides a language where code coverage property can be verified which is not possible with extension based object definition.

It has been mentioned that subtyping properties are based on extension declaration in the previous section. While variant is an opposite approach where the principal type refers other types such subtyping properties are not provided by extensions. Then subtyping variants are compared directly implying in this case a structural subtyping. Such subtyping property is valid if and only if the effective type is a subset of the required type.

2.1.3 The first class object `null`

In the language there is no special treatment linked to `null` value. Unlike C or Java^{TM} languages an attribute cannot be initialized with `null`. The reason came from the availability to modify the value of `this` in a class. Therefor all the possible values must be specified and then the special case dedicated to `null` as been removed.

Nevertheless the `Nullable` entity concept has been designed and proposed implying a default object provided in the language: `null`. In this case such object is clearly defined and typed.

Examples

```
package clump.lang

final object Null {}

type Nullable<E> = Null | E

final object NullPointerException extends RuntimeException {}

final interface INullable<E> {
    boolean isNull();
    E getOrDefault(E);
    E get() throws NullPointerException;
    E get<T>(T) throws T;
}

final class nullable<E>(Nullable<E>) implements INullable<E> {
    // Implementation ...
}

/* Basic package entities */

final Null null = new Null()

final E getFromNullable<E>(Nullable<E> n) {
    return nullable<E>(n).get();
}

final boolean isNull(Nullable<void> n) {
    return nullable<void>(n).isNull();
}
```

Therefor a denoted object can be set to `null` if and only if it’s specified.

Chapter 3

Compilation and Clump

3.1 Required system and tools

The following installation is valid with the version 0.1. Referenced scripts are designed for unix like systems. On windows the `java` command can be used as specified in this document.

Building the Clump system requires two softwares. First one was Java^{TM} SDK 1.5 or upper. Version 1.4 and lesser are not supported because of Generics and iterators. Second software required is ant (http://ant.apache.org/) for the xml based build engine.

3.2 Building the Clump compiler

Examples

```
> cd <CLUMP>
> ant -f compiler.xml

| Buildfile: compiler.xml
|
| clean:
|
| build:
|     [jflex] Generated: GenLex.java
|     [javac] Compiling 268 source files to <CLUMP>/trunk/classes
|     [javac] Note: Some input files use unchecked or unsafe operations.
|     [javac] Note: Recompile with -Xlint:unchecked for details.
|     [javac] Compiling 2 source files to <CLUMP>/trunk/classes
|
| archive:
|       [jar] Building jar: <CLUMP>/lib/Clump.jar
|
| compiler:
|
| BUILD SUCCESSFUL
| Total time: 3 seconds
```

3.3 Checking the compiler

The previous command produces one jar in the `<CLUMP>/lib` directory named Clump.jar.

Examples

```
> cd <CLUMP>
> java -jar lib/Clump.jar

| usage: java -jar lib/Clump.jar [compile|execute] [arguments]
```

This command can be executed using an unapropriate Java^{TM} version. In this case a specific message is printed notifying the Java^{TM} version incompatibility.

Examples

```
> java -version

| java version "1.4.2_18"
| Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_18-b08-314)
| Java HotSpot(TM) Client VM (build 1.4.2-90, mixed mode)

> cd <CLUMP>
> java -jar lib/Clump.jar

| # Abort: Unsupported virtual machine version 1.4.2-90 (must be 1.5.0 or upper).
```

3.4 Compiling the Clump library

This can be done using different modes. The the shortest one is the execution of the following command.

Examples

```
> cd <CLUMP>
> ant -f clump.xml library

<compilation traces>
```

or it can also be an explicit call using java.

Examples

```
> cd <CLUMP>
> java -jar lib/Clump.jar compile -o site/bin site/src
```

or simply calling the script clumpc located in the `<CLUMP>/bin` directory.

Examples

```
> cd <CLUMP>
> ./bin/clumpc -o site/bin site/src
```

If no output directory is specified all compilation stages are done except the code generation and its definitive compilation. If the Java^{TM}version is 1.5 or upper. If it's only the runtime - and not a complete SDK - an error is raised during the compilation.

Examples

```
> cd <CLUMP>
> ./bin/clumpc -o site/bin site/src
Compiler not found in .../1.5.0/Home (Its not a java SDK)
```

3.5 Executing the regression tests

A first set of regresion tests is provided. These tests can be executed with the following command.

Examples

```
> cd <CLUMP>
> ant -f clump.xml test

Buildfile: clump.xml

| tests:
|      [java] The args attribute is deprecated. Please use nested arg elements.
|      [java] Checking Exceptions
|      [java] [TEST> Test exception [1] succeed
|      [java] Checking Booleans
|      [java] [TEST> Test booleans [1] succeed
<...>
|      [java] [TEST> Test generic methods [2] succeed
|      [java] Checking native method
|      [java] [TEST> Test Native code [1] succeed
|
| BUILD SUCCESSFUL
| Total time: 0 seconds
```

or it can also be an explicit call using java.

Examples

```
> cd <CLUMP>
> java -jar lib/Clump.jar execute -l site/bin test.execution.all.main
or
> java -jar lib/Clump.jar execute -l lib/clump-library.jar test.execution.all.main

| Checking Exceptions
| [TEST> Test exception [1] succeed
| Checking Booleans
| [TEST> Test booleans [1] succeed
<...>
| [TEST> Test generic methods [2] succeed
| Checking native method
| [TEST> Test Native code [1] succeed
```

or simply calling the script `clump` located in the `<CLUMP>/bin` directory.

Examples

```
> cd <CLUMP>
> ./bin/clump -l site/bin test.execution.all.main
or
> ./bin/clump -l lib/clump-library.jar test.execution.all.main
```

3.6 Compiling and executing Clump source code

From now end-user can write its own source code. Compiling this code can be easily done using [-l URL] specifying library for the compilation. In fact the compiler generates corresponding java source code and Clump abstract object for future compilation. Such -l can be used more than once in the command line specifying each time a library to be used during the compilation stage.

Examples

```
> cd <CLUMP>
> ./bin/clumpc -o <OUTDIR> -l site/bin -l <ADDITIONAL_DIR> <SRC>
or
> ./bin/clumpc -o <OUTDIR> -l lib/clump-library.jar -l <ADDITIONAL_DIR> <SRC>
```

Executing such compiled source code is also done specifying libraries to be used. Therefor a specific class can be executed with the following command.

Examples

```
> cd <CLUMP>
> ./bin/clump -l <OUTDIR> -l site/bin -l <ADDITIONAL_DIR> <CLASS> [args]
or
> ./bin/clump -o <OUTDIR> -l lib/clump-library.jar -l <ADDITIONAL_DIR> <CLASS> [args]
```

As mentioned in the previous paragraph library can be referenced using an URL. It means if a java archive is built and provided by a specific Web site it can be referenced using [-l http://my.web.site/.../MyClumpLibrary.jar]