Generating Order-Sorted Data Types in JAVA

Project Report

by

Jan Schäfer

February 5, 2004
# Implementation

4.1 Type Hierarchy .............................................. 24
4.2 Sorts .......................................................... 25
  4.2.1 Implementation of the is Function .................... 27
  4.2.2 The KatjaSort class .................................... 28
4.3 Katja Built-in Types ...................................... 29
  4.3.1 nil ......................................................... 31
4.4 Term Productions .......................................... 31
  4.4.1 Tuples .................................................... 32
  4.4.2 Lists ..................................................... 34
  4.4.3 Variants ................................................ 39
4.5 Selectors .................................................... 39
  4.5.1 Implementation of Selectors ........................... 40
  4.5.2 Variant Productions and Selectors .................... 40

# The Katja program

5.1 Requirements .............................................. 43
5.2 Implementation Process .................................. 43
5.3 Program Components ..................................... 43
  5.3.1 The Scanner ............................................ 44
  5.3.2 The Parser ............................................. 45
  5.3.3 The Katja Specification of Katja ....................... 45
  5.3.4 The Analyser .......................................... 46
  5.3.5 The Generator ......................................... 47
5.4 Usage ........................................................ 47
  5.4.1 Requirements ........................................... 47
  5.4.2 Preparations ............................................ 48
  5.4.3 Run Katja ................................................ 48
  5.4.4 Command Line Parameters ............................. 49

# Conclusions and Future Work

6.1 Conclusions .............................................. 51
6.2 Future Work ............................................... 51

# References

A Migration from MAX to Katja

  A.1 The MAX system .......................................... 53
  A.2 Differences of the Specification Language ............... 53
  A.3 Differences of the JAVA Interface ....................... 54
  A.4 Case Study: The Formula Representation in Jive ....... 54
    A.4.1 The State Before the Migration ..................... 55
    A.4.2 The First Steps ..................................... 55
    A.4.3 Introducing Selectors ............................... 56
List of Figures

1. General structure of a Katja specification file ................. 7
2. One class for every term production .......................... 14
3. The classes KatjaTerm, KatjaNode and KatjaElement .......... 14
4. Built-in types ................................................. 15
5. The KatjaBuiltInType class and the type hierarchy .......... 15
6. The KatjaSort class and the type hierarchy ................... 16
7. All Katja elements realized as single classes .................. 17
8. The generated classes inherit from their production classes .. 17
9. Variant productions implemented as interfaces .................. 18
10. Variant productions and KatjaElement ......................... 19
11. The two type hierarchies ...................................... 24
12. The four main parts of Katja ................................. 44
13. JFlex generates two files and has one as input ............... 45
14. CUP has three files as input and one as output ............... 45
15. Katja generating files from katja.katja using the parser. 47
16. The Specification class and a selection of some other classes generated from the katja.katja file. 48
17. The generator classes ........................................ 50

List of Listings

1. A Katja specification file example ............................ 11
2. A simple Katja example ....................................... 16
3. Example how to iterate through a list with an iterator ...... 23
4. Implementation of the is function with a switch ............. 27
5. Implementation of the is function with the isInstance method 27
6. The KatjaElementImpl class .................................. 28
7. The KatjaSort class .......................................... 29
8. The KatjaBuiltInType class .................................... 30
9. The KatjaBool class ........................................... 30
10. The KatjaTerm interface ...................................... 32
11. The KatjaTuple interface ..................................... 32
12. The KatjaTermImpl class ..................................... 32
13. The KatjaTupleImpl class ..................................... 33
14. The T class .................................................. 33
15. The KatjaList interface ...................................... 35
16. The KatjaListImpl class ..................................... 36
17. The L class .................................................. 37
18. The Literator class .......................................... 38
19. The Katja specification of the Katja language .............. 45
1 Introduction

There are often situations that require a lot of different order-sorted data types. For example, in programming language specification and implementation many of them are needed after the parsing step to construct an abstract syntax tree. In Java you would normally have to write the classes for the data types manually. This can be very time consuming and error-prone.

In my project report I present a system which takes a specification file with the definition of order-sorted data types as input and generates the according Java classes from it. The specification file is much smaller, easier to read, to understand and to maintain than the hand-written Java source code. It is especially useful for programming language specification and implementation, but it can be used in many other situations that require order-sorted data types.

We call the system Katja, which stands for Kaiserslautern attribution system for Java.

There is already a similar system which produces C-code instead of Java code. It is called MAX\(^1\) [5], and it is also a system to support programming language specification and implementation. The specification language of Katja is derived from MAX, but it is slightly different, as it introduces selectors, which I explain later.

In this work I only implement a subset of the full specification language, namely the term representation. In later works Katja can be extended to support the full language, which will include a node representation and a functional programming language.

Overview

This document is structured as follows. Section 2 describes the Katja specification language. It describes the structure of a Katja specification file, the specification of order-sorted data types with term productions, and how to use external Java classes within a Katja specification.

Section 3 discusses the Java interface of Katja. It describes the Java classes that are included in Katja, and those which are generated by Katja from a specification file.

The implementation details of the Java classes are shown in Section 4.

Section 5 describes the Katja program itself and its different components. It also shows how Katja was used for the implementation of Katja and it describes the usage of Katja.

Finally, Section 6 gives conclusions and future work.

There are also two Appendices. Appendix A describes how to migrate from MAX to Katja. Appendix B contains the whole source code of Katja.

\(^1\)MAX stands for "Munich Attribution system for UNIX"
2 The Katja Specification Language

In this section I describe the Katja specification language.

The Katja specification language is derived from the MAX specification language. MAX was influenced by C and borrows the C-preprocessor. I removed all C-specific syntax and replaced it by syntax constructs that are more Java-like. The syntax also had to be adapted because of the selectors which are a new feature in Katja.

2.1 General Structure

A Katja specification file ends with .katja. Its general structure is shown in Figure 1.

| package name | import statements | term productions |

Figure 1: General structure of a Katja specification file

The file begins, like in Java, with a package name, which is followed by a list of import statements. Below that, the term productions are given.

2.1.1 Package Name

The first line contains the package name. The generated Java files will be in a package of the same name. It is declared like follows:

```
package packagename
```

The line starts with the package keyword, which is followed by the name of the package. Note that there is no semicolon at the end of the line. Like in Java, this line can be left out if the classes should be in the default package.

2.1.2 Import Statements

The line with the package name is followed by lines containing the import statements. There are two kinds of import statements:

- Java import statements
- Katja import statements
**Java import statements**  With a JAVA import statement JAVA classes can be declared. These classes can be classes from the JAVA-API or user defined classes. **Katja** does not check the existence of the JAVA classes, but if they do not exist, the JAVA source code generated by **Katja** will not be compilable, as the JAVA compiler will expect them.

JAVA import statements are defined like in Java:

```
import the.package.name.classname
```

The line starts with the `import` keyword, followed by the full package name of the class and the class name. It is similar to JAVA, except that the statement is not terminated with a semicolon.

Note that it is not possible to import whole packages with an asterisk as class name like in Java!

**Katja import statements**  With **Katja** import statements other **Katja** specification files can be imported. So it is possible to modularize the **Katja** specification and divide it into several files. **Katja** will parse and check the imported file, and all sorts declared in the imported file can be used as if they were declared inside the importing file. **Katja** files can be imported recursively, that is, **Katja** files that are imported by imported files are imported, too. However, cyclic or duplicate imports are not possible. **Katja** import statements are defined as follows:

```
import path/of/the/imported/file.katja
```

The line starts with the `import` keyword, followed by the path of the imported file. The path can be absolute or relative to the importing file. It is important that the filename has the `.katja` file extension!

### 2.2 Term Productions

With term productions data types are defined. There are three different kinds of term productions:

- Tuple production
- List production
- Variant production

They will be explained in the following.
2.2.1 Tuple Production

A tuple holds a fixed number of components, which have to be defined at creation time. The subcomponents of a tuple instance can neither be changed nor deleted. The only way to get a different tuple instance is to create a completely new one. Thus the interface is functional.

The definition of a tuple production looks as follows:

tuplename ( sort1, sort2, ... , sortn )

The line is started with the name of the tuple. The subcomponents are surrounded by round braces and separated by commas.

With this syntax, the subcomponents of a tuple could only be accessed by their exact position. In addition to this simple syntax KATJA supports selectors.

Selectors are names to access subcomponents, without knowing the exact position of the subcomponent inside the tuple. Code that uses selectors instead of absolut positions is much more readable, understandable and maintainable than accessing the subcomponents by their absolute position (see Section 4.5 for an example).

The name of a selector is written behind the subcomponent identifier:

tuplename ( sort1 sel1, sort2 sel2, ... , sortn seln )

This notation is borrowed from JAVA, where the variable name follows the variable type in the same way. Another notation would have been possible, by separating the sort identifier from the selector with a colon and to use blanks to separate the sort identifiers. With this notation the migration from MAX to KATJA would be easier, but JAVA programmers had to get used to a new syntax. So I decided to use the JAVA-like syntax.

Of course not all subcomponents need to have a selector, although it is highly recommended.

2.2.2 List Production

A list is initially empty and can hold an arbitrary number of elements of one sort\textsuperscript{2}. The order of the elements within the list is fix and cannot be changed.

The interface of a list sort is functional as it is for tuples. Adding or removing elements result in a new list, rather than in the original one being changed.

The definition of a list production looks like follows:

\textsuperscript{2}It is possible, however, to add elements that are a subsort of the one sort, so actually it is possible to add an arbitrary number of sorts, as long as they have a common super-sort
listname * sort

The name of the list is written first, followed by an asterisk and followed by a sort identifier.

2.2.3 Variant Production

A variant can be seen as a "union" of sorts. It is similar to a super-class in JAVA. That is, everywhere where a variant sort is used all sorts that are contained in that variant can be used in its place.

The definition of a variant production looks like follows:

\[ \text{variantname} = \text{sort1} \mid \text{sort2} \mid \ldots \mid \text{sortn} \]

The name of the variant is written first, followed by an equal sign, followed by a list of sort names and separated by the bar character.

2.2.4 Nodes

Every term can be transformed to a corresponding node and vice versa. A node is quite different from a term. A node knows its parent and its siblings. A node can also have additional attributes and context conditions. Nodes are beyond the scope of this work and I do not describe them further. It is, however, important to know that KATJA will be extended by nodes in the future, so that it can be designed accordingly.

2.3 Built-in Types

There are a number of built-in types that can be used in KATJA:

- Int
- Char
- String
- Bool

Int is an integer that has the same definition range like the JAVA int. Char is a single character. String is a string of characters. Bool is a boolean sort.

As described in the next section, every JAVA class can be used in a KATJA specification. So instead of the built-in types, it is also possible to use the JAVA wrapper classes e.g. java.lang.Integer. Then, however, the KATJA sort mechanism cannot be used and it is not possible to use the switch statement over the sort constants.

Built-in types can be used in lists or tuples, but they cannot be used in variants!
2.4 Using Java Classes

External JAVA classes can be used in a KATJA specification file. To use a JAVA class it has to be imported first as described in Section 2.1.2. All JAVA classes of the java.lang package are predefined and they do not have to be imported.

JAVA primitive types like int, char or boolean cannot be used in the specification file!

JAVA classes can only be used in lists or tuples, but cannot be used directly in variants!

2.5 Comments

Comments are written like in JAVA. A comment can be indicated by /* */ and text behind two slashes // will be treated as a comment until the end of the line. JavaDoc (/** */) comments will also be treated as normal comments.

2.6 Example

In Listing 1, a KATJA specification file example is shown.

/*
 * K a t j a s p e c i f i c a t i o n f i l e e x a m p l e
 */
package example
import java.util.Hashtable

// A tuple containing a Java class
SampleTuple ( Hashtable names )

// A list containing a predefined sort
SampleList * Int

// A variant
SampleVariant = SampleTuple | SampleList

Listing 1: A KATJA specification file example

2.7 Functions

Data types are useless if there are no functions defined on them. The KATJA specification language defines a set of functions on the different types of KATJA sorts. In this project report I only implement the term representation of the KATJA language, but not the full functional KATJA language. The functions presented below, however, occur in the JAVA interface of KATJA.

In this section I summarize the functions that are defined on terms.
2.7.1 General Functions

Every KATJA element has a sort, so the sort related functions are defined on all KATJA elements:

- **Sort sort(Element e)**
  Returns the sort of e.

- **Bool is(Element e, Sort s)**
  Indicates whether e is of the sort s.

There is also a function to test for the equality of two elements:

- **Bool eq(Element e1, Element e2)**
  Returns whether e1 and e2 are equal.

2.7.2 Term Functions

The following functions are defined on terms. They both work with tuples and lists.

- **Element subterm(Int ith, Term t)**
  Returns the ith subterm of t.

- **Int numsubterms(Term t)**
  Returns the number of subterms of t.

2.7.3 List Functions

A list has some additional functions:

- **Element first(List l)**
  Returns the first element of l.

- **Element last(List l)**
  Returns the last element of l.

- **List front(List l)**
  Returns l without its last element.

- **List back(List l)**
  Returns l without its first element.

- **List appfront(Element e, List l)**
  Appends e to the front of l.

- **List appback(Element e, List l)**
  Appends e to the end of l.

- **List conc(List l1, List l2)**
  Concatenates l1 and l2 and returns the resulting list.
3 The Java Interface

In this section I discuss the JAVA interface of KATJA. The interface consists of two parts: The JAVA classes that are generated by KATJA and the classes that KATJA already contains and which are used by the generated classes.

The design decisions of the JAVA interface are partly connected with the implementation of the JAVA classes. However, I discuss interface and implementation separately. The implementation is presented in Section 4.

A number of decisions have to be made for the JAVA interface: Which classes should be generated? What should the type hierarchy look like? Which methods should be in which class? These questions and their solutions are discussed in the following sections.

3.1 Requirements

Before I discuss the JAVA interface, I first present the requirements that the interface has to fulfill:

1. The interface should be as object-oriented as possible, thus JAVA programmers, used to objects, can use KATJA in an intuitive way.
2. All methods have to be functional because side-effects are not allowed.
3. The interface should be type-safe.
4. The interface should be designed with the background of extending it later by nodes.
5. It should be possible to use self-written JAVA classes inside the KATJA specification.
6. The generated code should be modular, that is two separately generated code bunches can be used together within the same program.
7. The generated code should be as fast as possible and use as few memory as possible.

3.2 Type Hierarchy

In this section I describe the type hierarchy of the JAVA interface. First I explain the classes that are provided by KATJA. These are classes for the term productions, the built-in types and the sorts. After this I present the interface of the classes that are generated by KATJA.
3.2.1 Term Productions

There are three different term productions: tuple, list and variant. Every production should have its own class (see Figure 2).

I said in Section 2.2.4 that Katja will be extended by nodes later. So nodes and terms have to be distinguished somehow. This should be done by the type hierarchy.

There should be a class for terms and for nodes, and a class which is the super-class of terms and nodes. I call the classes KatjaTerm, KatjaNode and KatjaElement (see Figure 3).

3.2.2 Built-in Types

There are a number of built-in types in Katja. For every built-in type there should be one class, and a super-class for all built-in types is needed (see Figure 4).

But should the built-in types be differenced by terms and nodes? The answer is yes, as built-in types can be either terms or nodes.
So the `KatjaBuiltInType` class has to inherit from `KatjaTerm` (see Figure 5).

Again, there will be a corresponding node class in later implementations.

### 3.2.3 Sorts

Every `Katja` element has a sort.

The question is, whether an additional type mechanism should be implemented, or whether the Java built-in type system can be used instead.

The answer is that a type mechanism beside the Java type system is needed, because it should be possible to do a switch over sort identifiers. This is not possible with the Java type system.

So I introduce a `KatjaSort` class for sorts. A `Katja` sort, however, is
also a Katja element, so in the type hierarchy KatjaSort inherits from KatjaElement (see Figure 6).

![Figure 6: The KatjaSort class and the type hierarchy](image)

### 3.2.4 Generated Classes

So far, I described the classes that are independent of a certain specification file. These classes are not being generated by Katja, but are already contained in Katja. Now, I describe the classes that are generated.

I use the Katja code example of Listing 2 to illustrate the different design decisions.

```java
T ( L | )
L * C
C = T | L
D = T
```

Listing 2: A simple Katja example

In this example there is a tuple T which only has the component L, accessible with the selector 1. L is a list of elements of the sort C. C is a variant production, and it can either be T or L. D is a variant production, too and it has sort T as subsort.

Which Java classes should be generated from Listing 2?

There are two general approaches: Generating a class for each production, that is, there would be a class T, L, C and D, or only create instances of the three production types tuple, list and variant.

With the second approach, a factory class [2] would be generated. This class would contain a static method for every sort that is defined in the specification file, which would create an instance of the corresponding sort.

This approach can easily be discarded by looking at requirement 3 (see Section 3.1). It says that the interface should be type-safe. Type-safety, however, can only be achieved by creating one class for every element.

Therefore, we chose the first approach. (see Figure 7).

Every class inherits from the class of their production class (see Figure 8).
3.2.5 Variant Productions

You may have already wondered if it is really necessary to have an extra class for a variant production. The answer depends on the implementation of variants. How should variants be implemented? A variant is a "union" of sorts, it is like a super-class in JAVA. So the straightforward way would be to implement variants as super-classes.

But there is a problem with this implementation: JAVA does not support multiple inheritance. At first glance this does not seem to be a problem. But let us look again at the initial example, especially at the last two lines:

\[
C = T \mid L \\
D = T
\]

If variants would be implemented with super-classes, \( C \) would be a super-class of \( T \) and \( L \), and \( D \) be a super-class of \( T \).

As in JAVA multiple inheritance is not allowed, and \( T \) would have to inherit from two classes, this approach does not work. But in JAVA it is possible for a class to implement two interfaces. So \( C \) and \( D \) could be interfaces and \( T \) would implement both of them.
Thus variants have to be interfaces, and elements of the variants have to implement these interfaces. (see Figure 9)

![Figure 9: Variant productions implemented as interfaces](image)

Now we have got a new problem. We have decided above that every Katja class has to inherit from KatjaElement. If this would be true, a variant had to inherit from KatjaElement. But then KatjaElement had to be an interface, because in JAVA an interface cannot inherit from a class, and we have seen that a variant has to be an interface.

Generally it would not be a big problem if KatjaElement had to be an interface, because there cannot be an instance of an abstract element. But as I explain in the implementation discussion later (Section 4), it would be better if KatjaElement could be an abstract class rather than an interface, because this avoids duplicated code in subclasses.

I decided to do both. The KatjaElement is an interface and every Katja class implements that interface. Additionally there is a class KatjaElementImpl from which KatjaTuple and KatjaList inherit, but variants do not. This works because there is no way to create an instance of an interface, and all methods of KatjaElement are implemented by the classes of the variant in any case.

KatjaElementImpl also implements KatjaElement, such that classes which inherit from KatjaElementImpl indirectly implement KatjaElement (see Figure 10).

3.3 Methods

So far, I have discussed which classes are generated by Katja, and what their type hierarchy looks like. Now I describe which methods are defined in the different classes.

3.3.1 Where Should the Methods Be Declared?

There are various functions that can be called on Katja elements (see Section 2.7). There are general functions that are defined on all elements, and some are only defined on terms, tuples or lists. There are also functions to convert JAVA types into Katja built-in types and vice versa. Where should
these functions be declared? Only in the class where they belong? Should, for example, the `front()` method only be declared in the `KatjaList` class? Or should it be already declared in one of its super-classes: `KatjaTerm` or `KatjaElement`?

In MAX, for example, it is possible to call all methods on every element, as MAX has no type-safety at compile time (it will produce a runtime error, however). But the JAVA interface of Katja should be type-safe, as said in the requirements (see Section 3.1). To achieve type-safety the methods should be declared in the classes where they belong.

But should `KatjaList` already contain all list related methods? Perhaps not, because then the `first()` method, for example, could only return `Object`. If, on the other hand, the `first()` method would be defined in the generated special list class (for example `L` in Listing 2), its specific type could be returned (C in the example).

So I decided that all methods are defined in the classes where the most type information is available, and these are the specific generated classes.

### 3.3.2 KatjaElement

The `KatjaElement` interface contains methods that have to be implemented by all `Katja` elements. These methods are:

- **public KatjaSort sort()**
  - Returns the sort of the element.

- **public boolean is(KatjaSort s)**
  - Returns whether the element is of the sort `s`. The method also returns true if `s` is a variant and contains the sort of the current object.
• public boolean eq(KatjaElement e)
  Returns whether the element is equal to the element e.

  Additionally the following Object methods are needed, to be able to use
  KATJA elements in other JAVA classes e.g. java.util.Hashtable:

• public boolean equals(Object o)
• public int hashCode()

  The toString method has to be implemented to get a meaningful output
  for KATJA elements:

• public String toString()

  In addition, every KATJA element class has the following static attributes:

• public final static int sortInt
  Stores the sort constant.

• public final static KatjaSort sort
  Stores the KatjaSort instance of the KATJA element.

  Where these attributes come from is explained in Section 4.2.

3.3.3 KatjaSort

The KatjaSort class implements all KatjaElement methods as it inherits
from this interface. In addition the following methods are defined:

• public int intValue()
  Returns the sort constant.

• public int toInt()
  For convenience reasons only, does the same as intValue.

  The intValue method, in combination with the static sortInt attribute, can be used to do a switch over different sorts.

3.3.4 KatjaTerm

The KatjaTerm interface contains only two methods:

• public int numSubterms()
  Returns the number of subterms.
• public int size()
  For convenience reasons only, does the same as numSubterms().

You may wonder why the subterm method is not defined in the KatjaTerm interface.
That is a result of a statement I made before: All methods are defined in the classes where the most type information is available. The subterm method is also defined on lists and lists only contain elements of one type. If the subterm method would be defined in the KatjaTerm interface already, the return type had to be Object and the type information of the lists would get lost.

3.3.5 KatjaTuple
The KatjaTuple interface also contains only two methods:

• public Object subterm(int ith)
  Returns the ith subterm.

• public Object get(int ith)
  For convenience reasons only, does the same as subterm.

Some questions could come to one’s mind: Why does the subterm method return Object and not KatjaElement, and why is it defined in the KatjaTuple interface already and not only in the generated tuple classes?

The subterm method returns an Object instead of a KatjaElement because of the requirement that it should be possible to use self-written JAVA classes inside the Katja specification. That is that subterms of a KatjaTuple could be either generated Katja classes or any other JAVA class. For this reason the subterm method returns Object.

The answer to the second question is that tuples do not have to contain subterms of the same type. They can contain any mixture of types. It would be possible, however, to check whether a tuple only contains one type and to use Object as return type only if it contains more than one type. But then the tuple interface would not be consistent. So I decided that the subterm method of tuples always returns Object.

KatjaTuple does not define all tuple methods. The generated tuples define the constructor and the selector methods. The methods of the class T generated from Listing 2, for example, look as follows:

• public T(L arg0)
  This is the constructor which takes one argument of type L.

• L l()
  This is the selector method to get the subterm l.
3.3.6 KatjaList

The KatjaList interface does not define any methods, as all methods contain type information, which would get lost if the methods were defined in the KatjaList interface already. All methods are only defined in the generated list classes. The methods of the class L generated from Listing 2, for example, look as follows.

- **public C subterm(int ith)**
  Returns the ith element of the list.

- **public C get(int ith)**
  For convenience reasons only, does the same as subterm.

- **public C first()**
  Returns the first element of the list.

- **public C last()**
  Returns the last element of the list.

- **public L front()**
  Returns a new list L without the last element.

- **public L back()**
  Returns a new list L without the first element.

- **public L appFront(C e)**
  Appends element e to the front of the list.

- **public L appBack(C e)**
  Appends element e to the back of the list.

- **public L add(C e)**
  For convenience reasons only, does the same as appBack.

- **public L conc(L l)**
  Returns a new list L that is a concatenation of this list and list l.

- **public L addAll(L l)**
  For convenience reasons only, does the same as conc.

- **public boolean contains(C e)**
  Whether the list contains the element e.

- **public void remove(C e)**
  Returns a new list L that is equal to the original one, but without the first occurrence of e. The first occurrence is determined by iterating through the list, starting with the first element and testing each element for equality with e using the equals method.
• public void removeAll(C e)
  Returns a new list L without any elements that are equal to e.

• public LIterator iterator()
  Returns an iterator for the list.

You might have noticed that I have extended the list interface described in Section 2.7.3. I have done this to make the list classes more conveniently useable, and to make them similar to the java.util.List interface that JAVA programmers are used to. Especially the iterator() method is interesting here, as it does not return a general Iterator class but a special one. These iterator classes are described in the next section.

3.3.7 List Iterators

Normally, in functional languages a list is iterated by recursively calling the front and back methods. This is still possible with the current list interface. But I added another way to iterate over lists, which is faster and easier to understand by people which are used to JAVA. I added an iterator method to every generated list class, which returns an object of a special iterator class. This class is called <ListName>Iterator and is generated for every list class.

The LIterator, for example, has the following methods:

• public boolean hasNext()
  Returns whether there is another element.

• public C next()
  Returns the next element.

With these iterators a list can easily be iterated with a while loop (see Listing 3 for an example).

```java
// ... Code in which the list l is created and filled
LIterator it = l.iterator();
while (it.hasNext()) {
  C c = it.next();
  // ... Do something with c
}
```

Listing 3: Example how to iterate through a list with an iterator
4 Implementation

So far I have described the Katja specification language and the Java interface of the provided and generated classes. In this section I go into the implementation details of the classes presented in the latter section. I start with the implementation of the sorts and the built-in types of Katja. After that the different term productions are described. Finally I discuss the implementation of the selectors.

4.1 Type Hierarchy

In Section 3.2 you have seen the Java interface type hierarchy. Additionally a second hierarchy exists (see Figure 11):

```
KatjaElement <-KatjaElementImpl
          <-KatjaTerm <-KatjaTermImpl
                      <-KatjaList <-KatjaListImpl
                      <-KatjaTuple <-KatjaTupleImpl
```

Figure 11: The two type hierarchies

Every class of the second hierarchy has a corresponding interface of the first hierarchy, which it implements. The partition was needed, because KatjaElement had to be an interface (see Section 3.2.5). Without the second class hierarchy, however, there would have been duplicated code in the generated classes. For example the is method, as described below, would have been implemented in every generated class.
4.2 Sorts

There are two sort related functions in Katja:

- Sort sort(Element e)
- Bool is(Element e, Sort s)

The first one returns the Sort of e, the second one checks whether e is of Sort s and returns a Katja Bool. The is function is similar to the instanceof operator in Java: It also returns true for variants of that type.

How should sorts be implemented? Should there be a separate class for every sort, or should there only be instances of the KatjaSort class?

I think it is not necessary to create a separate class for every sort. A sort does not change over time and must only store one attribute to distinguish the different sorts.

But of what kind should this attribute be? One important point of the sort implementation is that it must be possible to do a switch over several sorts. This is a feature that is not possible with the JAVA type system.

A switch statement in JAVA expects a char, byte, short or an int. I think an integer fits best for this purpose. To have the biggest range of numbers I decided to use int.

There is another important thing to know: The case statements of the switch statement expect constant values which have to be known at compile time. This is a big problem. Imagine there are two bunches of separately generated code, and they should be used together. The sort constants must be known at compile time, and the sort constants of both code bunches are fixed. The KATJA system did not know of the other code when it generated the sort constants. But if both generated code bunches are to be used together, the sort constants have to be unique, otherwise it would not be possible to do a switch over two sorts with the same sort constant.

It is not possible to find a unique integer for every sort name because the possible number of sort names is larger than $2^{32}$. Even if the length of the sort names would be restricted to 8 characters there are $2^8 = 208,827,064,576 \approx 209$ Billion possibilities versus $2^{32} = 4,294,967,296 \approx 4.3$ Billion different JAVA ints. So there is always an opportunity that two sorts will get the same sort constant.

There are two general ways of generating the sort constants: Writing a hash function that takes the sort name as input and produces a hash value from it, or generating the sort contants randomly.

The first approach has the advantage that the sort constant of a sort is always the same. This could be, however, also be a disadvantage as in the case of two equal sort constants it would not be adequate to regenerate the sort constants as they would be identical again.

A good hash function would choose the constants evenly spread over the whole interval. But as the sort constants are used for switch statements in
Java, this is a disadvantage because the switch statement is faster if the case constants differ not too much [4]. In addition, the probability of two equal sort constants is the same as generating random sort constants.

So I decided to generate the sort constants randomly. But I do not pick a random sort constant for every sort, but only for the first one. The other constants are then calculated by incrementing from the first one.

As I use ints for the sort constants, there are $2^{31}$ different positive numbers. The built-in types are not generated, so they can have predefined numbers. I decided to reserve 1-999 for them. When the code is generated a random positive integer from the interval $[1, 2^{31}/1000] = [1, 2147483]$ is picked. This number is then multiplied by 1000 and the sort constants are calculated by incrementing from this number.

Besides that the sort constants are now close to each other and though the switch statement is faster, this has another advantage: The probability that two separately generated bunches of code have the same sort constant is lower than picking a random number for every sort constant. To be precise, the probability is $1:2147483$ (assuming that both Katja specifications have 1000 sorts each). If each sort constant would be taken randomly from the whole interval $[1, 2^{31}]$, instead of incrementing it, the probability of two identical sort constants is $1:2147$ (same assumption as above).

Proof: Let the number of sorts of both code bunches be 1000. Without loss of generality let the sorts of the first code bunch have the numbers 1000, 1001, ..., 1999. For every sort of the second code bunch a random number of the interval $[1000, 2^{31}]$ is picked. The probability of one sort to hit one of the numbers of the first code bunch is about $1:2147483 \approx 0.000000466$. The probability that at least one sort of the second code bunch has the same number as a sort of the first code bunch is the same as 1 minus the probability that no sorts at all have the same sort numbers. That is $1 - (1 - 0.000000466)^{1000} = 1 - 0.999534447 = 0.000465553 \approx 1 : 2147$.

In the rare case that there are two identical sort constants, and they are both used in the same switch statement, the Java compiler would complain about it and would not compile the code. In that case, one of the code bunches has to be generated again.

But there are some problems with this approach. The first one is that it is not 100% sure that two separately generated code bunches have unique sort constants. The second one is that if the code is generated from the same Katja file two times, the generated files will differ from each other. This might be a problem, for example when the generated code is managed by version control systems like CVS.

\footnote{You might argue that generated code should not be added to a version control system, as it is sufficient to add the Katja file, for example, from which the code is generated. This would, however, require the users to have Katja installed on all systems where the code might be used.}
So I decided to give the user the opportunity to set the starting integer for the sort constants with a command line argument (see Section 5.4.4).

### 4.2.1 Implementation of the is Function

How can the is function be implemented? As I said above, every KATJA sort has an integer as attribute, so it is possible to do a switch over all sort integers. The is function of the T class from Listing 2 could then look like the example given in Listing 4.

```java
public class T implements KatjaTuple {
    public static final int sortInt = 101;
    public static final KatjaSort sort =
        new KatjaSort(101);

    boolean is(KatjaSort s) {
        switch (s.toInt()) {
            case KatjaElement.sortInt:
            case KatjaTerm.sortInt:
            case KatjaTuple.sortInt:
            case T.sortInt:
            case C.sortInt:
            case D.sortInt: return true;
            default: return false;
        }
    }
    // ...
}
```

Listing 4: Implementation of the is function with a switch

In larger examples this implementation can lead to long switch statements resulting in a lot of code.

As every KATJA element has its own class, it is possible to use the JAVA built-in type information for the KATJA sorts. So the is function can be implemented with the JAVA instanceof operator. The problem is that the instanceof operator needs a static type as right-hand-side operand. The Class class of JAVA has a function that emulates the instanceof operator and takes a Class object as parameter.

I decided to give the Sort class an additional attribute that holds a Class object and to implement the is function by using the isInstance method of the Class class. The is function of the T class from Listing 2 could then look like in Listing 5.

```java
public class T implements KatjaTuple ,C,D {
    public static final int sortInt = 101;
    public static final KatjaSort sort =
        new KatjaSort(T.class,101);
```
boolean is(KatjaSort s) {
    return s.getClassValue().isInstance(this);
}
// ...
they are always identical, and as the attribute is static there is only one additional integer per class and not per object.

The KatjaSort class looks as follows:

```java
public final class KatjaSort extends KatjaElementImpl {
    public final static int sortInt = 5;
    public final static KatjaSort sort =
        new KatjaSort(KatjaSort.class, 5);

    private Class cl;
    private int id;

    protected Class getClassValue () {
        return cl;
    }

    public KatjaSort(Class cl, int id) {
        this.cl = cl;
        this.id = id;
    }

    public KatjaSort sort() {
        return KatjaSort.sort;
    }

    public boolean eq(KatjaElement s) {
        if (! (s instanceof KatjaSort))
            return false;
        return this.id == ((KatjaSort) s).id;
    }

    public String toString () {
        return cl.getName();
    }

    public int intValue () {
        return id;
    }

    public int toInt () {
        return id;
    }

    public int hashCode () {
        return id;
    }
}
```

Listing 7: The KatjaSort class

4.3 Katja Built-in Types

The built-in types are very easy to implement. There are no real alternatives for their implementation.

The classes of the built-in types are only wrappers around JAVA built-in types. I illustrate the implementation by means of the KATJA Bool type. The class KatjaBool, however, is a little special among the built-in types,
as a boolean type can only have two values. It would be an overhead to always create a new object for the same value. Therefore the constructor of KatjaBool is private and there exist only two instances of the KatjaBool class, which can be accessed by static final attributes of the class. There is also the factory method \[2\] fromBool that returns a KatjaBool instance from a JAVA boolean parameter. This results in only having two instances of the KatjaBool class throughout the whole program. The KatjaBool class is shown in Listing 9. All classes of the built-in types inherit from KatjaBuiltInType class, which is shown in Listing 8.

```java
public abstract class KatjaBuiltInType
    extends KatjaElementImpl
    implements KatjaTerm {

    public final static int sortInt = 6;
    public final static KatjaSort sort =
        new KatjaSort(KatjaBuiltInType.class,6);

    public int numSubterms() {
        return 0;
    }
    public int size() {
        return 0;
    }
}
```

Listing 8: The KatjaBuiltInType class

```java
public class KatjaBool extends KatjaBuiltInType {
    public final static int sortInt = 10;
    public final static KatjaSort sort =
        new KatjaSort(KatjaBool.class,10);

    public static final KatjaBool TRUE = new KatjaBool(true);
    public static final KatjaBool FALSE = new KatjaBool(false);

    boolean value;

    private KatjaBool(boolean value) {
        this.value = value;
    }

    public static KatjaBool fromBoolean(boolean b) {
        return b ? TRUE : FALSE;
    }

    public boolean booleanValue() {
        return value;
    }

    public boolean eq(KatjaElement e) {
```
Listing 9: The KatjaBool class

The other types (Int, Char, String) are all implemented similarly. They have an attribute that holds a value of a Java built-in type (int, char, String), and they contain methods which return that attribute (toInt, toChar, toString). To create an instance of a built-in type, the constructor has to be called with the corresponding Java type. All implementations are functional, so internal values of Katja built-in types cannot be changed, but only new instances can be created.

4.3.1 nil

Special attention has to be given to nil. nil is returned by a Katja list if an item of an empty list is accessed.

There are some possibilities of the nil implementation:

- Create an extra class or interface KatjaNil and let all Katja classes inherit from it.
- Using the null keyword of Java

The first approach does not work, because it should be possible to use a Java class as an element of a Katja list which is not generated by Katja. With the first approach every Java class had to inherit from KatjaNil which would be too restrictive.

So we decided to simply use null for nil.

4.4 Term Productions

Now we have sorts and built-in types and are ready to create more complex types: tuples, lists and variants.

The discussion below uses the Katja example from Listing 2:

\[
\begin{align*}
T & ( L \ 1 ) \\
L & * C \\
C & = T \ | \ L \\
D & = T
\end{align*}
\]
4.4.1 Tuples

Tuples have a fixed number of subterms and the sorts of the subterms are also fixed. There are two functions that are dealing with tuples: `subterm` and `numSubterms`.

One possibility to implement tuples is to use an array to store the subterms. An array has a fixed size, and the implementations of both methods are one-liners and have a complexity of $O(1)$.

But there is another way to implement them. As the code is generated, and the number of subterms of a tuple is fixed, it would be no problem to create an attribute for every subterm of the tuple. With this realization the reference to the array can be saved.

So I decided to create an attribute for every subterm. The `KatjaTuple` interface only contains the `subterm` method. The `numSubterms` method is already defined in the `KatjaTerm` interface. The `KatjaTupleImpl` class has no methods at present. I added it nevertheless to be prepared for future changes.

The source code of the interfaces can be seen in Listing 10 and Listing 11. The classes `KatjaTermImpl` and `KatjaTupleImpl` are shown in Listing 12 and 13. The source code of the T class that is generated from Listing 2 can be seen in Listing 14.

```java
public interface KatjaTerm extends KatjaElement {
    public final static int sortInt = 2;
    public final static KatjaSort sort = new KatjaSort(KatjaTerm.class, 2);

    public int numSubterms();
    public int size();
}

Listing 10: The KatjaTerm interface
```

```java
public interface KatjaTuple extends KatjaTerm {
    public final static int sortInt = 3;
    public final static KatjaSort sort = new KatjaSort(KatjaList.class, 3);

    public Object subterm(int ith);
    public Object get(int ith);
}

Listing 11: The KatjaTuple interface
```

```java
public abstract class KatjaTermImpl extends KatjaElementImpl implements KatjaTerm
```

32
public int size() {
    return numSubterms();
}

Listing 12: The KatjaTermImpl class

class KatjaTupleImpl extends KatjaTermImpl implements KatjaTuple
{
    public final int sortInt = 1950446000;
    public final KatjaSort sort = new KatjaSort(T.class, 1950446000);

    private L _child0;

    public T(L arg0) {
        this._child0 = arg0;
    }

    public L l() {
        return _child0;
    }

    public int numSubterms() {
        return 1;
    }

    public Object subterm(int ith) {
        switch (ith) {
            case 0: return _child0;
            default: throw new KatjaIllegalArgumentException(
                "Trying to access subterm " + ith + " but the sort " + sort().toString() + " has only 1 subterm!");
        }
    }

    public boolean eq(KatjaElement e) {
        if (! e.is(T.sort))
            return false;

        T t = (T) e;

    }

Listing 13: The KatjaTupleImpl class
if (! t._child0.equals(_child0))
    return false;
return true;
}
public int hashCode() {
    return 1950446000 +
              _child0.hashCode() * 31;
}
public KatjaSort sort() {
    return T.sort;
}
}

Listing 14: The T class

4.4.2 Lists

A list is initially empty, and the number of subterms is not fixed. A list can only contain elements of one sort.

The functions that are defined on lists are the ones that are defined on tuples namely subterm and numSubterms, and the additional list functions front, back, first, last, appFront, appBack and conc.

How should the list be implemented? Generally, there are two possibilities: Using an array or a linked list. In Java there are two Collection classes for this: ArrayList and LinkedList. Both have their advantages and disadvantages.

First there are differences in the performance of the methods: An ArrayList is fast when accessing a random element by the subterm method, but is slow when adding an element to the front with the appFront method. The appBack method is fast as long as the capacity of the array is large enough, otherwise the array has to be resized which takes some time. A LinkedList is fast when adding new elements to the front or the back, but is slow when accessing a random element.

What about the space consumption? The LinkedList class has an extra class for every element, containing the element itself and references to the previous and next element. The ArrayList class, however, holds some extra space at the end of the array to be able to quickly add elements to the end, without resizing the array every time.

Concerning speed, the LinkedList class fits better for our purpose, because the subterm method will be seldomly used on a random position. Instead a list will be iterated from the beginning until the end.

But the discussion above has not considered an important point yet: The implementation of the Katja list has to be functional. This makes most arguments obsolete, because for every operation that changes the list, a complete new list has to be created, and all data has to be copied to the
new list. In that discipline, however, \texttt{ArrayList} is much faster than the \texttt{LinkedList}, because \texttt{LinkedList} has to create a new object for each of its entries.

The methods \texttt{front} and \texttt{back} can be implemented without replicating the whole list, because they do not introduce new objects. In my first implementation I decided to use the \texttt{ArrayList} class of \texttt{Java}, but in the future it can be replaced by an own implementation to deal with the special requirement of a functional interface. Then it would be possible to implement the \texttt{appBack} and \texttt{appFront} methods without copying the whole list in some cases. The \texttt{front} and \texttt{back} methods can be efficiently implemented by the \texttt{subList} method of the \texttt{ArrayList} class, as it does not copy the whole list.

I also decided to slightly extend the list interface. First I added some methods that do the same as the ones that are already mentioned, but which have names that are borrowed from the \texttt{JAVA List} interface, which \texttt{Java} programmers are used to. These methods are: \texttt{get} instead of \texttt{subterm}, \texttt{size} instead of \texttt{numSubterms}, \texttt{add} instead of \texttt{appBack} and \texttt{addAll} instead of \texttt{conc}. Additionally I added some methods for convenience reasons: \texttt{remove(Object o)} that removes the first occurrence of the object \texttt{o}, \texttt{removeAll(Object o)} that removes all occurrences of the object \texttt{o}, and \texttt{contains(Object o)} which returns \texttt{true} if the object \texttt{o} is contained in the list and \texttt{false} otherwise.

Further I added a method \texttt{iterator()} that returns an iterator object to iterate over the whole list. The iterator class is called \texttt{<ListName>Iterator} and is generated additionally to the list class.

I created an interface called \texttt{KatjaList} and a class called \texttt{KatjaListImpl} which implements the interface. \texttt{Katja} list productions inherit from the \texttt{KatjaListImpl} class, so that it is easy to change the list implementation later.

The \texttt{KatjaList} interface has no methods, as the \texttt{numSubterms} and \texttt{size} methods are already declared in the \texttt{KatjaTerm} interface, and all other methods have specific type information in their signature and cannot be declared at this hierarchy level (see Listing 15).

```java
public interface KatjaList extends KatjaTerm{
    public final static int sortInt = 4;
    public final static KatjaSort sort =
        new KatjaSort(KatjaList.class,4);
}
```

Listing 15: The \texttt{KatjaList} interface

The current \texttt{KatjaListImpl} implementation is just a wrapper around the \texttt{ArrayList} class of \texttt{Java}. An important difference, however, is that instead of throwing an \texttt{IndexOutOfBoundsException}, \texttt{null} is returned when a non-existent element is accessed. The source code can be seen in Listing 16.
package katja.common;

import java.util.ArrayList;
import java.util.List;

public abstract class KatjaListImpl extends KatjaElementImpl implements KatjaTerm {

    protected List values = new ArrayList();

    protected abstract KatjaListImpl createInstance(List values);

    protected Object subtermInternal(int ith) {
        if (ith < 0)
            throw new KatjaIllegalArgumentException(
                    "Trying to access subterm "
                    + ith + " but only values greater or equal to 0 are allowed!");

        if (ith >= values.size())
            return null;

        return values.get(ith);
    }

    public int numSubterms() {
        return values.size();
    }

    protected boolean containsInternal(Object o) {
        return values.contains(o);
    }

    protected Object firstInternal() {
        if (values.size() == 0)
            return null;

        return values.get(0);
    }

    protected KatjaListImpl frontInternal() {
        if (values.size() == 0)
            return null;

        return createInstance(values.subList(0, values.size() - 1));
    }

    protected KatjaListImpl appFrontInternal(Object o) {
        List list = new ArrayList(values.size() + 1);
        list.add(o);
        list.addAll(values);

        return createInstance(list);
    }
}
```java
protected KatjaListImpl removeInternal(Object o) {
    if (values.size() == 0)
        return null;

    List list = new ArrayList(values);
    list.remove(o);

    return createInstance(list);
}

protected KatjaListImpl removeAllInternal(KatjaListImpl l) {
    if (values.size() == 0)
        return null;

    List list = new ArrayList(values);
    list.removeAll(l.values);

    return createInstance(list);
}

protected KatjaListImpl concInternal(KatjaListImpl l) {
    List list = new ArrayList(values.size() + l.values.size());
    list.addAll(values);
    list.addAll(l.values);

    return createInstance(list);
}

public boolean eq(KatjaElement e) {
    if (! (e instanceof KatjaListImpl))
        return false;

    KatjaListImpl el = (KatjaListImpl) e;
    return el.values.equals(values);
}

public int hashCode() {
    return values.hashCode() +
        this.getClass().hashCode() * 31;
}
```

Listing 16: The KatjaListImpl class

The L class that is generated from Listing 2 simply calls the methods of KatjaListImpl ending with 'Internal' and casts the result to its specific type C. The method parameters are also of the specific type (see Listing 17).

```java
import Katja.common.*;
import java.util.List;

public class L extends KatjaListImpl implements C {
```
public static final int sortInt = 897491001;
public static final KatjaSort sort =
    new KatjaSort(L.class,897491001);

public C subterm(int ith) {
    return (C) subtermInternal(ith);
}
public C first() {
    return (C) firstInternal();
}
public L appFront(C e) {
    return (L) appFrontInternal(e);
}
public LIterator iterator() {
    return new LIterator(values.iterator());
}
protected KatjaListImpl createInstance(List values) {
    L l = new L();
    l.values = values;
    return l;
}
public KatjaSort sort() {
    return L.sort;
}
// ...

Listing 17: The L class

The LIterator class is just a wrapper around the java.util.Iterator
which returns the correct type instead of Object (see Listing 18).

import java.util.Iterator;

public class LIterator {
    private Iterator iterator;
    public LIterator(Iterator it) {
        this.iterator = it;
    }
    public boolean hasNext() {
        return iterator.hasNext();
    }
    public C next() {
        return (C) iterator.next();
    }
}

Listing 18: The LIterator class
4.4.3 Variants

In Section 3.2.5, I stated that variants are implemented with interfaces. This decision was influenced by the implementation of variants.

If variants are implemented with interfaces, and the interfaces are implemented by the members of variants, it is possible to check if an element is in a variant by simply using the `instanceof` operator of `JAVA`.

But there is another way of implementing variants: Having no interfaces at all. As we are generating the `JAVA` code it would be no problem to do a `switch` over all possible elements of a variant to check if an element is in that variant. This could be even faster than the `instanceof` operator.

This approach has some problems. The `KatjaSort` implementation that I describe in Section 4.2 is only possible if interfaces are used for variants. And there is another point which leads to interfaces. Imagine you have generated the `JAVA` code from a `KATJA` specification, and you want to write a `JAVA` method which takes a certain variant as parameter. If there would be no interface for it, the type of the parameter had to be of `KatjaElement`, and there would be no type-safety. Thus the type has to be checked at runtime.

These arguments led to my decision to use interfaces for variant productions. As you will see in Section 4.5.2, these interfaces can contain selector methods depending on their subsorts.

4.5 Selectors

One problem of the existing MAX system [5] is that subterms of tuples can only be selected by their absolut position. This has some disadvantages:

- It is not very readable, as the subterm position gives no clue about the purpose of the subterm.
- It is error-prone, as the subterm positions are easily mixed up.
- It is not well maintainable, as tuples cannot easily be extended with new subterms, other than appending them to the back.

These arguments led to the introduction of selectors. Selectors are just names for subterms of tuples. Take the following example:

```java
BooleanExpression ( Expression, BooleanOperator, Expression )
```

If, for example, the variable `b` is a `BooleanExpression` and you would like to access the `BooleanOperator` of the `BooleanExpression` you have to write `b.subterm(1)`. If someone reads this code, one could not say what this method returns without knowing the exact definition of the `BooleanExpression` tuple. And if the position of the `BooleanOperator` is
to be changed in the BooleanFormula tuple, all subterm calls have to be found and adopted accordingly.

With selectors the following can be written:

BooleanExpression ( Expression leftExpr,
                      BooleanOperator operator,
                      Expression rightExpr)

The BooleanOperator can be accessed by using the selector method operator(). You can write b.operator(), which is more readable, maintainable and easier to remember, which makes it less error-prone.

You might have noticed that the tuple subterms are separated by commas instead of blanks compared to MAX. The syntax of the tuple definition had to be changed because of the selectors. Otherwise it could not be differed whether a tuple with two names in it is a tuple with two subterms or whether it is a tuple with one subterm and a selector for this subterm.

4.5.1 Implementation of Selectors

The implementation of the selectors is straightforward. For every selector one method with the name of the selector is generated. This method simply returns the corresponding subterm.

```java
public class BooleanExpression ... {

  public BooleanOperator operator () {
    return _child1;
  }

}
```

The use of selectors has another advantage: The selector method returns the correct type of the subterm. If the subterm method is used to access a subterm of a tuple, a JAVA Object is returned. To execute any method on the Object, a cast has to be done in advance. If a selector is used to access the subterm instead, no cast is needed, because an object of the corresponding type is returned. This makes the code even more readable and less error-prone.

4.5.2 Variant Productions and Selectors

What if there are two tuples which have the same selectors and both are part of a variant production? You might expect the selector to be accessible in the variant, too. I explain that by means of the following example⁴:

```
4This example is taken from the KATJA file of the KATJA language
```
Production  = TupleProd | VariantProd | ListProd

TupleProd  ( SortId sortId,TupleParamList params,
             Int sortInt )

VariantProd ( SortId sortId,SortIdList params,
               Int sortInt )

ListProd   ( SortId sortId,SortId param,Int sortInt )

All three tuples have the selector sortId and sortInt and all selectors have the same type, so it would be no problem to let the variant have these selectors, too.

But what is about the other selectors?
The tuples TupleProd and VariantProd both have a selector called params, but the type is different. We cannot add two methods with the same name and different return types to the variant interface, because it is not allowed in JAVA to do this. But what if we return a supertype of both return types? TupleParamList and SortIdList are both list productions. So we could add a method called params() to the variant Production and let it return a KatjaList object. But this would not work either, because the params() method is also defined in the classes TupleProd and VariantProd, but with a different return type. Both classes implement the Production interface, because they are part of that variant. So they must both implement the params() method with the KatjaList return type, too. This is, however, not possible, because both of them already have a method called params() with its specific return type.

One solution would be to have only the method params() with the KatjaList return type. But then a cast would have to be done each time this method is used. The better solution is to not add the params() method to the variant interface. If now the params() method is used on an object of the static type Production, a cast to the specific type has to be done before the method call.

In the discussion above I left out an important point: The ListProd tuple does not have a params selector at all, but a selector called param. I have already explained that the Production interface will not have a params() method because of the different return types. But what is about selectors which do not occur in all members of a variant, but which all have the same return type? The param selector only exists once and so has always the same return type. Should this selector be available in the Production class? Perhaps yes. In that case the other classes which implement the Production interface would have to implement the param() method, too. The implementation of these methods would be in throwing an IllegalOperationException or in returning null.

The problem with that approach is that a variant production could have many different methods, which all belong to different tuple productions. In that case all other tuple productions of the variant must implement these
methods. This would result in a lot of code and would be very confusing, because there are many methods that do not do anything other than throwing an Exception or returning null. In my opinion this is not worth the advantage of having the selectors already in the variant production. So I decided to put only a selector in a variant interface if all tuples of the variant have the same selector, that is the same name and the same type and otherwise not. There might be, however, situations in which there are many subsorts of a variant production which have the same selector, and only a few or only one do not have it. In this case it would be nice to let the user decide if the selector should be put into the variant. So in the future the syntax of Katja could be extended to allow the definition of virtual selectors on variants. That is, all subsorts of the variant must have that selector even if the selector method would do nothing useful.

What if a variant contains a list production? A list cannot have a selector, because a list is initially empty and it would make no sense to give a list subterm a selector. So in the case that a variant contains a list production no selectors at all are put into the variant interface. The explanation is the same as above. All selectors would be put into the list classes with methods which would do nothing useful and that would be confusing.

If the variant contains other variants, the selectors of these variants are put into the variant if all other productions of the variant also contain the same selectors.
5 The Katja program

So far, I have described which Java classes are provided by Katja and which are generated by Katja from a specification file. This section describes how they are generated, that is, I first describe the Katja program itself and then I describe how to work with Katja.

First I summarize the requirements for the Katja program. Then I describe the program and the main parts.

5.1 Requirements

Katja has to be written in Java. It has to read a Katja specification file and check the file for syntactic and semantic errors. If the specification file is free of errors, the corresponding Java classes have to be generated.

5.2 Implementation Process

I implemented the Katja program step-by-step. I started with the smallest running program. The first version did not support all language features such as selectors and did not have a semantic analyser, but it was possible to generate Java classes from a Katja specification file.

Now I did a boot-strapping step. I wrote a Katja specification file for the Katja language and let Katja generate the corresponding Java classes. After that, I adapted Katja to use the generated Java classes. That means that the parser returns a KatjaTerm object after the it has finished the parsing and that the other program components work with this KatjaTerm object.

In the next step I extended Katja to be able to handle selectors. I updated the Katja specification file and introduced selectors. Then I re-generated the classes from the specification file and adapted the Katja program to use the selectors instead of the absolute element positions.

As the Katja specification language is not a full programming language, the boot-strapping had to end at this point. Boot-strapping has to be done again when the katja.katja file changes. In the future there could be additional boot-strapping steps when nodes are added to Katja or when Katja features a full functional programming language.

Finally I wrote a semantic analyser to check various error conditions. Each program component is described in more detail in the following.

5.3 Program Components

The Katja program consists of four main components:

- A scanner which scans the Katja file.
- A parser which takes the output of the scanner and parses it.
• A analyser which takes the output of the parser and analyses it for semantic errors.

• A generator which generates the JAVA classes.

The scanner and parser parse the Katja file and return a Specification object. The analyser takes the Specification object, which is a Katja term, and does the semantic analysis. If no errors have been detected the generator takes the Specification object and generates the JAVA files (see Figure 12).

![Figure 12: The four main parts of Katja](image)

5.3.1 The Scanner

The scanner is automatically generated by JFlex [3] from the file katja.lex. JFlex is a scanner generator written in JAVA which generates JAVA files. It would have been possible to use the lex file that is used in MAX, as JFlex has a compatibility mode for lex files. I preferred to write the lex file from scratch, so that it was possible to use all features of JFlex.

Before parsing a specification file, MAX uses a C pre-processor to remove comments, apply #define statements and to include other files with the #include statement. I avoided the use of a pre-processor for the following reasons: Comments can be handled by the scanner, #define statements are C-specific and JAVA developers are not used to them, besides they can be replaced by methods. The #include statement is also very C-specific and I preferred to use the import keyword instead. I also wanted to be able to import JAVA classes which would not be possible by including a JAVA file.

---

5 Actually that is only true if there are no #ifdef statements. But #ifdef statements make no sense in a MAX file, so that #define statements can be replaced by methods.
The scanner generates two files: `Scanner.java` which contains the actual scanner and `Symbols.java` which contains the symbol constants for the parser (see Figure 13).

![Figure 13: JFlex generates two files and has one as input.](image)

### 5.3.2 The Parser

The parser is also automatically generated. It is generated by CUP [1] from the file `katja.cup`. It uses the files `Scanner.java` and `Symbols.java` which are generated by JFlex (see Figure 14). The parser also uses classes generated by Katja itself to build up a KATJA term of the specification. Besides the `Specification` object, the parser fills a hash table with all production names. If there are two productions with the same name, an error is thrown and the parser stops. However, the parser does not check the definition of types used within productions, this is done by the analyser.

![Figure 14: CUP has three files as input and one as output.](image)

### 5.3.3 The Katja Specification of Katja

To have a first test of the KATJA system, I decided to do boot-strapping and use classes generated by KATJA within KATJA. For this I wrote a KATJA specification of the KATJA language. The file is called `katja.katja` and the content can be seen in Listing 19.

```java
package katja.spec

Specification ( PackageDcl pck,
  ImportList importList,
```
ProductionList prodList, 
String filename)
PackageDecl = PackageName | Empty
PackageName ( String name)
ImportList * Import
Import = JavaImport | KatjaImport
JavaImport = JavaSimpleImport
| JavaOnDemandImport
JavaSimpleImport ( String name, Int line )
JavaOnDemandImport ( String name, Int line )
KatjaImport ( String name, Int line )
ProductionList * Production
Production = TupleProd | VariantProd | ListProd
TupleProd ( SortId sortId,
TupleParamList params,
Int sortInt )
TupleParamList * TupleParam
TupleParam = SortId | Selector
Selector ( SortId sortId, String selector )
VariantProd ( SortId sortId, SortIdList params,
Int sortInt )
ListProd ( SortId sortId, SortId param,
Int sortInt )
SortIdList * SortId
SortId ( String name, Int line )
Empty ( )
/*
 * Helper classes
*/
SelectorList * Selector
KatjaError ( String message,
String file,
Int line )
KatjaErrorList * KatjaError
KatjaWarning ( String message,
String file,
Int line )
KatjaWarningList * KatjaWarning

Listing 19: The Katja specification of the Katja language

Katja takes the katja.katja file as input, using the parser that is generated by CUP, and generates the corresponding Java files to the spec directory (see Figure 15). The generated Specification class contains all information about the parsed Katja file (see Figure 16).

5.3.4 The Analyser

The analyser checks the Specification object returned by the parser for semantic errors. Currently it checks the absence of the following situations:
Figure 15: **Katja** generating files from `katja.katja` using the parser.

- A name which is neither defined by a Java import nor by a term production is used in a production.
- A Java primitive type is used in a production.
- A Katja built-in type or a Java class is used in a variant production.
- There is a cyclic dependency between two or more variant productions.

5.3.5 The Generator

The generator takes the `Specification` object and generates the Java classes from it. The generator consists of a couple of classes. The main class is the `PackageGenerator` class which takes the `Specification` object and creates the package directory. It then creates a `ClassGenerator` object for every `Production` that is contained in the `Specification` object and calls its `generate` method. For every production type there is a specific generator class that generates its Java file (see Figure 17).

5.4 Usage

In this section I describe how to work with Katja.

5.4.1 Requirements

Katja is written in Java so it is platform independent and runs on all operating systems where a Java virtual machine is installed. It is delivered as one JAR file, which is called `katja.jar`. To run Katja the following programs have to be installed:

- At least JRE 1.3
- JFlex [3]
- Java CUP [1]

---

6 Java Archive
7 The JRE can be downloaded from [http://java.sun.com](http://java.sun.com)
5.4.2 Preparations

In order to run KATJA the JFlex classes, the JAVA CUP classes and the katja.jar file have to be added to the CLASSPATH environment variable.

5.4.3 Run Katja

KATJA is a command line program, that is, it has no GUI\(^8\). To run KATJA open a console and type

```
java katja.Katja [options] <inputfile>
```

where `<inputfile>` is a KATJA specification file and `[options]` is a list of command line parameters. Without any options KATJA will generate the JAVA classes to the current working directory.

\(^8\)Graphical User Interface
5.4.4 Command Line Parameters

To change the behavior of KATJA a number of command line parameters can be used:

- `-h  -help`
  Show a help message.

- `-d=<dir>  -dest=<dir>`
  Generate the JAVA classes to `<dir>`. If this is not set, the classes are generated to the current working directory.

- `-q -quiet`
  Be quiet, suppress any output.

- `-debug`
  Show debug messages

- `-nogen`
  Only analyse, but do not generate anything.

- `-sortint=<n>`
  Start sort integer constants at `<n>`.
Figure 17: The generator classes
6 Conclusions and Future Work

In this project report I presented the Katja program, a system to generate order-sorted data types in Java. I described the Katja specification language and the Java interface of the generated Java classes as well as the implementation details. This work was the first step in the Katja development process, namely the implementation of the term representation.

6.1 Conclusions

The term representation proved to be quite useful already. I used it in the Katja program itself, by writing a Katja specification of the Katja language and used the generated Java classes in the analyser and the generator. This shows the usefulness of Katja in language specification and implementation.

Katja was also successfully applied in the Jive tool to represent boolean formulas, and in fact replaced the MAX system that was used before. In Appendix A I describe that migration process.

6.2 Future Work

This work was the first step in the Katja development process. In later works Katja will be extended by the node representation and a full functional programming language, which will be similar to the MAX programming language.

In a first step the node representation should be implemented. The node representation is much more useful for language specification and implementation than the term representation. The extension should be relatively easy. The scanner, parser and analyser can be left untouched. There only has to be found a Java interface for nodes and conversion methods from terms to nodes and vice versa. Also the generators for the node classes will have to be implemented. After this step, there could be a boot-strapping step, by partly rewriting the Katja program to use nodes instead of terms to have a first working example.

In a second step the full functional language can be implemented. This will be much more work. The scanner, the parser and the analyser have to be extended to support attributes, context conditions, methods and a set of expressions. A Java representation of the Katja language has to be found, and the generator classes for it have to be written.

Acknowledgments

I would like to thank Nicole Rauch and Prof. Dr. Arnd Poetzsch-Heffter who supported me with many comments and corrections.
References


A Migration from MAX to Katja

This section describes how to migrate from MAX to Katja. It describes the migration from a MAX specification file to a Katja specification file and the migration from the MAX JAVA interface to the Katja JAVA Interface, it does not explain the conversion of C functions to JAVA.

A.1 The MAX system

The MAX system [5] is a tool to support programming language specification and implementation. Like Katja it takes a specification file as input which contains a description of order-sorted datatypes. But MAX generates C code. As C is a pure procedural programming language it does not generate any classes. In addition to the C code a JAVA interface is generated. The JAVA interface uses the JNI\(^9\) to access the generated C code. This interface is described later in Section A.3. The full specification language of MAX features, in addition to the definition of order-sorted datatypes, the definition of attributes, context conditions and a full functional programming language. Katja only supports the definition of order-sorted datatypes yet, but in the future these features could be added.

A.2 Differences of the Specification Language

The Katja specification language is very similar to the MAX specification language, but there are some differences:

- Katja does not use a C pre-compiler, so you cannot use \#define or \#include statements. To import another Katja file you have to use the import keyword.
- You can import JAVA classes, which is not possible with MAX.
- Tuple subterms are separated by commas instead of blanks.
- You can add selectors to tuple subterms.
- In Katja you cannot use built-in types in a variant, which is possible in MAX.

These differences lead to the need of adapting the MAX specification files to the Katja syntax when performing a migration.

\(^9\)JAVA native interface
A.3 Differences of the Java Interface

The MAX Java interface is just a one-to-one mapping from the C interface to the JNI\(^\text{10}\). So the Java interface is just a bunch of \texttt{public static} methods defined in one class. The only type is a \texttt{Long}.

This realization has some disadvantages:

- The Java native interface is rather slow, as it has to convert the Java parameters of methods to C and vice versa.
- The interface uses no object-oriented features and is not intuitive for Java programmers.
- There is no use of advanced features of the Java virtual machine, like the garbage collector.
- The interface is not type safe, as every MAX element is just a Java \texttt{Long}.
- It is not possible to use two separately generated MAX libraries together.

The Katja interface is totally different. It consists of many classes, one class for every production, and the methods are in the classes where they belong to.

The Katja interface tries to use Java types as often as possible to avoid the call of conversion functions. Instead of using the MAX built-in types like \texttt{Bool} or \texttt{Int} the Java types \texttt{boolean} and \texttt{int} are used. The Katja \texttt{is} method for example returns a Java \texttt{boolean} instead of a MAX Bool. The call of a conversion function like \texttt{MAXetob} of the MAX interface is not necessary. This saves a lot of code and makes the code more readable.

The Katja interface is type-safe. For example it is not possible to call the \texttt{is} method with anything other than a \texttt{KatjaSort}. This makes the interface less error-prone and saves also many runtime checks.

It is not necessary to compile C code to a shared library and to add this library to the \texttt{LD_LIBRARY_PATH} variable, and there is no need of loading this library.

The only thing you have to do is to add the \texttt{katja.jar} file and the Katja-generated Java classes to your \texttt{CLASSPATH}, and you can use the generated files.

A.4 Case Study: The Formula Representation in Jive

In the following I will explain the migration of MAX to Katja by means of the the Jive system.

---

\(^\text{10}\)Java native interface
The Jive system is a tool to semi-automatically prove the correctness of programs written in a subset of Java. Jive is mainly written in Java, but uses the MAX system, among other things, to generate an abstract formula representation.

The migration from MAX to Katja was needed (and it was one of the reasons to write the Katja system at all), because the formulas could become so large that they exceeded the reserved memory of MAX, which crashes the MAX system. Also JNI is so slow that some calculations needed hours to be finished if the formulas became too large.

A.4.1 The State Before the Migration

There were two MAX files regarding the formulas. One with the definition of the formula productions (MAXFormula.m) and one with MAX functions operating on formulas (MAXFormulaFunctions.m). As at this time it was not possible to write functions in Katja, I had to rewrite all MAX functions in Java. This, however, turned out to be a great test for the new Java interface. There was also much code written in Java that uses the JNI of the MAX system, which had to be ported to the Katja Java interface, too.

Some questions came to my mind before starting the migration:

- Would the Java code of Katja be more complicated and larger than the MAX Java code?
- How easy would it be to convert the MAX code to Java and how long would it take?
- Is the type-safety of use, or would there be the need of many casting operations?
- Would the new language feature, the selectors, bring any advantages?
- And last but not least, would the problems of the MAX system with speed and memory usage be solved?

A.4.2 The First Steps

At first I renamed the file MAXFormula.m to Formula.katja. After that I adopted the content of the file to the new Katja syntax. That actually means that the tuple parameters have to be separated by commas instead of blanks. This is an important step, because it can happen that Katja does not complain about the syntax if there are only tuples with less than three children. In that case tuples with two children are interpreted as tuples with one child and a selector for it. Katja does not complain about this but generates code that would not be as expected.
After adding the commas between the tuple parameters I did a first test and started Katja with the changed file:

```java
java katja.Katja -nogen Formula.katja
```

The `-nogen` options let Katja only analyse the file and not generate anything. The result was that Katja aborted with the following error message:

```
Error in file Formula.katja:19:The variant 'identifier' contains the built-in type 'KatjaString', which is not allowed!
```

The line of the error looks as follows:

```java
identifier = String
```

The error means, that it is not allowed anymore to put a built-in type into a variant. The solution is to create a new tuple production, let this tuple production be part of the variant and let the new tuple production contain the built-in type. As the variant 'identifier' which Katja complains about only contains one element, I converted the variant to a tuple and let the only child be a `KatjaString`.

The changed line looks as follows:

```java
identifier ( String )
```

After fixing this bug and running Katja again it does not complain anymore and successfully analyses the file.

So the whole transformation of the MAX file to Katja syntax took me about 5 minutes.

### A.4.3 Introducing Selectors

But now I did not generate the code yet. It would be a pity if I would not use the new features of Katja, especially the selectors. So I overworked the Formula.katja file and added selectors to all tuples.

I changed, for example, the following line of code:

```java
fBinaryExpr ( fExpr, fBinaryOp, fExpr )
```

It describes a binary expression with two operands and an operator. I added selectors and the resulting line looks like follows:

```java
fBinaryExpr ( fExpr left, fBinaryOp op, fExpr right )
```
To get the operator of the binary expression, it now no longer necessary to remember that the operator was the 2nd child, but instead it can be accessed by the selector `op`.

After all tuple parameters had their selectors I added the package name of the package where the generated class should be in:

```plaintext
package jive.PVC.Container.Formula.Katja
```

Now the Katja file was finished and I let Katja generate the Java classes:

```plaintext
java katja.Katja jive/PVC/Container/Formula/Formula.katja
```

The files were generated as expected and I started to convert the MAX functions to Java.

### A.4.4 Conversion of MAX Functions to Java

All MAX functions were stored in a file called `MAXFormulaFunctions.m`. As all functions in the Java interface of MAX are static, I created a new class `FormulaFunctions` and put all translated functions in that class in a first step. Later I put the functions to the classes that they actually belong to.

Some functions could simply be deleted, because the new interface made them obsolete, like the following function:

```plaintext
/* returns 1st operand of a fBinaryExpr */
FCT getOperand1 (fExpr ebe) fExpr:
  IF is[ebe, fBinaryExpr]: st1(ebe)
  ELSE nil()
```

This function is in fact a selector! So I could replace all occurrences of `getOperand1` with the selector `left`.

Other functions were getting obsolete because of the methods that I added to the list interface for convenience reasons. For example the following function could be replaced by the `contains` method:

```plaintext
/* returns whether the fExprList el contains the fName ex. */
FCT containsVariable (fExprList el, fExpr ex) Bool:
  IF is[ex, fNameOrConstant]:
    IF ns(el) = 0: false()
    ELSE
      IF eq[st1(el), ex]: true()
      ELSE containsVariable(el.back, ex)
  ELSE dbgsecure("Error in containsVariable : 2nd param is of wrong sort", ex)
```

57
So I identified all functions that were obsolete and replaced all their occurrences by the new methods.

The translation of most of the functions was straightforward. For example the function:

```plaintext
/* returns whether the fExpr e is an implication */
FCT isImplication(fExpr e) Bool:
    IF numsubterms(e) = 3:
        IF is([subterm(2,e), _fImplies]: true())
            ELSE false()
        ELSE false()
```

looks in Java as follows:

```java
public boolean isImplication() {
    if (! expr.is(fBinaryExpr.sort))
        return false;
    if (((fBinaryExpr) expr).op().is(fImplies.sort))
        return true;
    else
        return false;
}
```

I replaced the `numsuberms(e) = 3` check with the actually correct sort check in this example. Also, I used the `op` selector instead of the `subterm` method.

In functions where lists were traversed with the `back` function, I used the `iterator` method instead:

```plaintext
/* returns whether the fExprList e is a sub-fExprList of el */
FCT isSubExprList(fExprList el, fExprList e) Bool:
    IF ns(e) = 0: true()
    ELSE
        IF isInExprList(el, st1(e)) = true():
            isSubExprList(el, e.back)
        ELSE false()
```

In Java:

```java
protected static boolean isSubExprList(fExprList el, 
    fExprList e) {
    fExprListIterator it = e.iterator();
    while (it.hasNext()) {
        if (! el.contains(it.next()))
            return false;
    }
    return true;
}
```
A.4.5 Conversion of MAX Java code to Katja Java code

After all MAX functions had been translated into JAVA I moved the methods to the classes where they belong to, mainly the Formula class and started to convert the MAX JAVA code to the KATJA JAVA code.

The Formula class stored a Long, that held a MAX fExpr term. With KATJA there exist a fExpr class now, so all Long occurrences had to be replaced by fExpr.

All MAX functions start in JAVA with MAX, this could be removed. Most conversion functions that convert from MAX built-in types to JAVA built-in types could be removed, because KATJA functions return JAVA built-in types directly. I also replaced all subterm function calls with the corresponding selector methods.

After I replaced the whole MAX related JAVA code, the migration was finished.

A.5 Conclusions

The results of the migration have been very encouraging:

- The JAVA code of KATJA was much more readable and understandable than the MAX JAVA code.

- The conversion of the MAX JAVA code to the KATJA JAVA code, and of the MAX code to JAVA were very easy and straightforward.

- The type-safety was of great use, and there was seldomly the need of casting operations.

- The selectors brought an enormous advantage to the readability and to the correctness of the code.

- During the migration I also found some bugs in the old code (Due to the bad readability of the MAX JAVA interface)
package katja;

import java_cup.runtime.Symbol;
import java.io.File;

%%% Scanner
%cup Sym
%line %column
%
StringBuffer string = new StringBuffer();
String filename;

public void setFilename(String filename) {
    this.filename=filename;
}

public String getFile(String filename) {
    System.out.println("Include Datei: "+filename+" ");
    if (filename.equals(""))
        return " ";
    if (filename.length() < 10)
        return " ";
    System.out.println(filename.length());
    filename = filename.substring(8).trim();
    System.out.println("Living1");
    if (filename.length() < 2)
        return " ";
    filename = filename.substring(1, filename.length()-1);
    System.out.println("Include Datei: "+filename+" ");
    System.out.println("Datei:\"+this.filename);
    File file = new File(this.filename);
    String sep = System.getProperty("file.separator");
    System.out.println("Living");
    return file.getParent()+sep+filename;
}

private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
}

private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
}

}%

LineTerminator = \r\n| \r
WhiteSpace = { LineTerminator } | [ \t \f ]
Comment = { TraditionalComment } | { EndOfLineComment } | { DocumentationComment }
TraditionalComment = " /s" [ ] "s/" { LineTerminator }*
EndOfLineComment = "//" { InputCharacter }* { LineTerminator }*
DocumentationComment = " /s" "s/" { LineTerminator }*
katja.cup

package katja;

import katja.spec.*;
import katja.common.*;
import java_cup.runtime.Symbol;

action code {
    void error(String message, int line, int column) {
        System.out.println("Syntax Error in file "+parser.filename+" line "+(line+1)+" column "+(column+1)+": "+message);
        parser.number_of_errors++;
    }
};

parser code {

String filename;
Specification spec;
Katja katja;

int number_of_errors = 0;

public int numberOfErrors() {
    return number_of_errors;
}

public Specification getSpecification() {
    return spec;
}

public void setKatja(Katja katja) {
    this.katja = katja;
    this.filename = katja.getSpecFile();
}

public void setFilename(String filename) {
```java
public void report_error(String message, Object info) {
    this.filename = filename;
}

public void report_error(String message, Object info) {
    // Create a StringBuffer called 'm' with the string 'Error' in it. */
    StringBuffer m = new StringBuffer("Error");
    m.append("in file ").append(filename).append(":");
    // Check if the information passed to the method is the same type as the type java_cup.runtime.Symbol. */
    if (info instanceof java_cup.runtime.Symbol) {
        // Declare a java_cup.runtime.Symbol object 's' with the information in the object info that is being typecasted as a java_cup.runtime.Symbol object. */
        java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
        // KatjaLexer lexer = (KatjaLexer) this.getScanner();
        // m.append(" regarding text: \"\"+lexer.yyytext()++\"\") ;
        // Check if the line number in the input is greater or equal to zero. */
        if (s.left >= 0) {
            // Add to the end of the StringBuffer error message
            // the line number of the error in the input. */
            m.append("line "+(s.left+1));
            // Check if the column number in the input is greater or equal to zero. */
            if (s.right >= 0) {
                // Add to the end of the StringBuffer error message
                // the column number of the error in the input. */
                m.append("column "+(s.right+1));
            }
        }
        // Add to the end of the StringBuffer error message created in this method the message that was passed into this method. */
        m.append(" :"+message);
        // Print the contents of the StringBuffer 'm', which contains an error message, out on a line. */
        katja.error(m.toString());
    }
    // Change the method report_fatal_error so when it reports a fatal error it will display the line and column number of where the fatal error occurred in the input as well as the reason for the fatal error which is passed into the method in the object 'message' and then exit. */

    public void report_fatal_error(String message, Object info) {
        done_parsing();
        report_error(message, info);
    }

    public void syntax_error(Symbol cur_token) {
        number_of_errors++;
        report_error("Syntax_error", cur_token);
    }

    public void unrecovered_syntax_error(Symbol cur_token) {
        katja.error("FatalError: Couldn’t repair and continue parse");
        done_parsing();
    }
}
```

62
terminal MULT, LBRACE, RBRACE, EQUAL, PIPE, COMMA, DOT;
terminal IMPORT, PACKAGE;
terminal java.lang.String IDENTIFIER;
non terminal Specification katja spec;
non terminal ProductionList production list;
non terminal Production production;
non terminal SortIdList pipe list;
non terminal TupleParamList tuple param list;
non terminal PackageDcl package declaration opt;
non terminal PackageName package declaration;
non terminal ImportList import declarations opt;
non terminal ImportList import declaration list;
non terminal Import import declaration;
non terminal Import single type import declaration;
non terminal Import type import on demand declaration;
non terminal KatjaString name;
non terminal KatjaString simple name;
non terminal KatjaString qualified name;
non terminal SortId sort id;
non terminal TupleProd tuple prod;
non terminal ListProd list prod;
non terminal VariantProd variant prod;
non terminal newlines;
katja spec ::= 
    package declaration opt:packagedcl
    import declarations opt:imports
    production list:productions
    |
    parser.spec = new Specification(packagedcl, imports, productions,new KatjaString(parser.filename));
    ;

package declaration opt ::= 
    package declaration:name
    |
    RESULT = name;
    ;

import declarations opt ::= 
    import declaration list: list
    |
    RESULT = list;
    ;

production list ::= 
    production: prod
{:
    ProductionList list = new ProductionList();
    RESULT = list.appBack(prod);
    :}
  :| production_list:list production:prod
  {:
    RESULT = list.appBack(prod);
  :}
  :
  |

import_declaration_list ::= import_declaratio
import_declaratio
{:
    ImportList importList = new ImportList();
    RESULT = importList.appBack(importdecl);
    :
    import_declaration_list:list import_declaratio
import_declaratio
{:
    RESULT = list.appBack(importdecl);
  :}
  ;

package_declaration ::= PACKAGE name:id
{:
    Katja.debug("Package declaration \"+id\" found \!");
    RESULT = new PackageName(id);
  :}

import_declaratio ::= single_type_import_declaratio
import_declaratio
{:
    RESULT = importdecl;
    :
    type_import_on_demand_declaratio
import_declaratio
{:
    RESULT = importdecl;
  :}
  ;

single_type_import_declaratio ::= IMPORT name:id
{:
    Katja.debug("Import declaration \"+id\" found \!");
    if (id.toString().endsWith(".katja"))
    RESULT = new KatjaImport(id,new KatjaInt(idLeft+1));
    else
    RESULT = new JavaSimpleImport(id,new KatjaInt(idLeft+1));
  :}

type_import_on_demand_declaratio ::= IMPORT name:id DOT MULT
{:
    Katja.debug("Import on demand \"+id\" found \!");
    RESULT = new JavaOnDemandImport(new KatjaString(id+".*"),
    new KatjaInt(idLeft+1));
  :}
IMPORT: im DOT MULT
{
  error("Missing import name after import keyword!", imleft +1, imright +1);
  RESULT = new JavaOnDemandImport(new KatjaString(""), new KatjaInt(imleft +1));
}

sort_id ::= name:id
{
  if (Katja.isReservedWord(id.toString())) {
    error("The identifier "+id+" is a reserved word in Java and cannot be used as Katja identifier!", idleft +1, idright +1);
  }
  if (Katja.isNativeType(id.toString())) {
    error("The identifier "+id+" is a Java native type and cannot be used as Katja identifier!", idleft +1, idright +1);
  }
  String s = Katja.getBuiltInName(id.toString());
  RESULT = new SortId(new KatjaString(s), new KatjaInt(idleft +1));
}

name ::= simple_name:id
{
  RESULT = id;
  }
  qualified_name:id
{
    RESULT = id;
  }

simple_name ::= IDENTIFIER:id
{
  RESULT = new KatjaString(id);
}

qualified_name ::= name:n DOT IDENTIFIER:id
{
  RESULT = new KatjaString(n+"."+id);
}

production ::= tuple_prod:prod
{
  RESULT = prod;
  }
  list_prod:prod
{
  
}
RESULT = prod;

| variant_prod:prod
| {
|   RESULT = prod;
| }

\[
tuple_{\text{prod}} ::= \text{sort}_{id} : \text{id} \text{ LBRACE } \text{tuple}_{\text{param list}} : \text{list} \text{ RBRACE}
\]

| {
|   Katja.debug("Tuple\text{Production} "+id.subterm(0)+" found!")
|   RESULT = new TupleProd(id, list, new KatjaInt(parser.katja.
|   getFreeSortInt()));
| }

| sort_id:id tuple_param list:list RBRACE
| {
|   error("Missing left bracket before tuple production parameters!", idleft+1,idright+1);
|   RESULT = new TupleProd(id, list, new KatjaInt(parser.katja.
|   getFreeSortInt()));
| }

\[
tuple_{\text{param list}} ::= \text{tuple}_{\text{param list}} : \text{list} \text{ COMMA } \text{tuple}_{\text{param}} : \text{param}
\]

| {
|   RESULT = list.appBack(param);
| }

| tuple_{param} : param
| {
|   TupleParamList list = new TupleParamList();
|   RESULT = list.appBack(param);
| }

\[
tuple_{\text{param}} ::= \text{sort}_{id} : \text{typeId} \text{ simple}_{name} : \text{sel}
\]

| {
|   RESULT = new Selector(typeId, sel);
| }

| sort_id:typeId
| {
|   RESULT = typeId;
| }

| sort_id:param name name
| {
|   error("Missing comma between tuple production parameters!", paramleft+1,paramright+1);
|   RESULT = param;
| }

| sort_id:id name:param name name
| {
|   error("Missing comma between tuple production parameters!", paramleft+1,paramright+1);
|   RESULT = id;
katja.katja

/*
 * This is the Katja specification of the Katja language.
 */

package katja.spec

Specification ( PackageDcl pck, ImportList importList, ProductionList prodList, String filename )
PackageDcl = PackageName | Empty
PackageName ( String name )
ImportList * Import
Import = JavaImport | KatjaImport
JavaImport = JavaSimpleImport | JavaOnDemandImport
JavaSimpleImport ( String name, Int line )
JavaOnDemandImport ( String name, Int line )
Katja Import ( String name, Int line )
ProductionList * Production
Production = TupleProd | VariantProd | ListProd
TupleProd ( SortId sortId, TupleParamList params, Int sortInt )
TupleParamList * TupleParam
TupleParam = SortId | Selector
Selector ( SortId sortId, String selector )
VariantProd ( SortId sortId, SortIdList params, Int sortInt )
ListProd ( SortId sortId, SortId param, Int sortInt )
Katja.java

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import katja.common.KatjaInt;
import katja.common.KatjaString;
import katja.spec.*;

/*
 * The main class of the Katja system
 * @author Jan Schäfer
 */
public class Katja {

    private static String VERSION = "0.1";
    
    private static boolean DEBUG = false;
    
    private static boolean QUIET = false;
    
    private static boolean NOGEN = false;
}
Maps built-in names to other Java names. E.g. String will be KatjaString, Int will be KatjaInt and so on

```java
private static HashMap builtInNames = new HashMap();

private static HashSet reservedWords = new HashSet();

private static HashSet nativeTypes = new HashSet();

static {
    builtInNames.put("String", "KatjaString");
    builtInNames.put("Int", "KatjaInt");
    builtInNames.put("Bool", "KatjaBool");
    builtInNames.put("Char", "KatjaChar");
    builtInNames.put("KatjaString", "KatjaString");
    builtInNames.put("KatjaInt", "KatjaInt");
    builtInNames.put("KatjaBool", "KatjaBool");
    builtInNames.put("KatjaChar", "KatjaChar");

    reservedWords.add("private");
    reservedWords.add("abstract");
    reservedWords.add("break");
    reservedWords.add("case");
    reservedWords.add("catch");
    reservedWords.add("class");
    reservedWords.add("const");
    reservedWords.add("continue");
    reservedWords.add("default");
    reservedWords.add("do");
    reservedWords.add("else");
    reservedWords.add("extends");
    reservedWords.add("final");
    reservedWords.add("finally");
    reservedWords.add("for");
    reservedWords.add("future");
    reservedWords.add("generic");
    reservedWords.add("goto");
    reservedWords.add("if");
    reservedWords.add("implements");
    reservedWords.add("import");
    reservedWords.add("inner");
    reservedWords.add("instanceof");
    reservedWords.add("interface");
    reservedWords.add("native");
    reservedWords.add("new");
    reservedWords.add("null");
    reservedWords.add("operator");
    reservedWords.add("outer");
    reservedWords.add("package");
    reservedWords.add("private");
    reservedWords.add("protected");
    reservedWords.add("public");
    reservedWords.add("rest");
    reservedWords.add("return");
    reservedWords.add("static");
```

69
```java
reserveWords.add("super");
reserveWords.add("switch");
reserveWords.add("synchronized");
reserveWords.add("this");
reserveWords.add("throw");
reserveWords.add("throws");
reserveWords.add("transient");
reserveWords.add("try");
reserveWords.add("var");
reserveWords.add("void");
reserveWords.add("volatile");
reserveWords.add("while");

nativeTypes.add("int");
nativeTypes.add("boolean");
nativeTypes.add("char");
nativeTypes.add("long");
nativeTypes.add("byte");
nativeTypes.add("double");
nativeTypes.add("float");
nativeTypes.add("short");

}/**
 * Contains all java imports.<br>
 * The keys are String objects with the classnames and the
 * values are String objects with the full qualified import.<br>
 * E.g.: for 'java.util.Iterator' the key would be
 * 'Iterator' and the value would be 'java.util.Iterator' <br>
 * All imports of all imported and the current Katja file are
 * stored.
 */
HashMap javaImportHash;

/**
 * Contains for every MAX SortId the package where
 * it is declared.<br>
 * The Keys and Values are String objects
 */
HashMap sortIdPackageHash;

/**
 * Stores the sort integer for the next sort.
 * non-internal types start with a random number
 * that is at least 1000 big, so that it doesn’t get into
 * the MAX built-in types sort integers.
 * <p>
 * The probability, that types of two different generated
 * MAX sorts have the same sortInt is then 1:2147482
 */
private int nextsortint = ((new Random()).nextInt(Integer.MAX_VALUE - 1000)/1000) * 1000;

/**
 * Contains the Specification term created by the KatjaParser
 */
private Specification spec;

/**
 * Stores for every SortId that is contained in
 * at least one Variant, a SortIdList which contains all
```
* Variants the SortId is contained in.
* Keys are the names of the SortIds
*/
HashMap variantHash;

/**
 * The directory where the classes should be generated
 */
String destDir = null;

/**
 * The name of the Katja specification file to parse
 */
String specFile = null;

/**
 * Contains the start time of Katja
*/
private long startmillis;

/**
 * Contains all error messages
 */
private KatjaErrorList katjaErrorList;
private KatjaWarningList katjaWarningList;

/**
 * A List of Katja instances of all
 * imported Katja files
 */
private List katjaImports;

/**
 * If this file is imported from another specification
 * this stores the Katja instance of that specification.
 * Otherwise it is &lt;code&gt;null&lt;/code&gt;
 */
private Katja parentKatja;

/**
 * Creates a new Katja instance with the given specification
 * file and a Katja parent instance.&lt;br&gt;
 * The destination directory will be taken from the
 * parent Katja instance.&lt;br&gt;
 * The nextsortint will be setted to the nextsortint of
 * the parent instance.&lt;br&gt;
 * A parent is a Katja instance which specification
 * file imports the specification of this instance.
 * @param specFile the specification file to parse
 * @param parentKatja the parent Katja instance which specification
 * file imports specFile
 */
public Katja(String specFile, Katja parentKatja) {
    this.specFile = specFile;
    this.destDir = parentKatja.destDir;
    this.nextsortint = parentKatja.nextsortint;
    this.javaImportHash = parentKatja.javaImportHash;
    this.sortIdPackageHash = parentKatja.sortIdPackageHash;
    this.variantHash = parentKatja.variantHash;
    this.parentKatja = parentKatja;
/**
 * Creates a new Katja instance with the given specification
 * file. The classes are generated to the current working directory.
 * @param specFile the path of the specification file
 * @throws Exception if an error occurs
 */
public Katja(String specFile) throws Exception {
    this(specFile,"") ;
}
/**
 * Creates a new Katja instance with the given specification
 * file. The classes are generated to the given destination directory. If destDir is empty or null
 * the current working directory is taken.
 * @param specFile the path of the specification file
 * @param destDir the directory where the classes should be generated to. If it is empty or null
 * the current working directory is taken
 * @throws Exception if an error occurs
 */
public Katja(String specFile, String destDir) throws Exception {
    destDir = _destDir ;
    specFile = _specFile ;
    if (destDir == null || destDir == "") {
        destDir = System.getProperty("user.dir");
    }
    javaImportHash = new HashMap();
    sortIdPackageHash = new HashMap();
    variantHash = new HashMap();
    katjaErrorList = new KatjaErrorList();
    katjaWarningList = new KatjaWarningList();
}
/**
 * Starts parses the specification file, analyses it and generates the classes
 * @throws FileNotFoundException if the specification file
 * or the destination directory doesn’t exist
 * @throws Exception if an error occurs
 */
public void parse() throws FileNotFoundException, Exception {
    startTimeMillis = System.currentTimeMillis();
    showCopyright();
    checkFileExistence();
    outputln("Specification file :"+this.specFile);
    outputln("Destination directory :"+this.destDir);
    output("Parsing ...");
    Parser parser = parseFile();
    if (parser.numberOfErrors() == 0) {
        outputln("done");
    }
}
293 output("Analysing... ");
294 fillVariantHashTable();
295 Analyser analyser = analyse();
296 if (analyser.numberOfErrors() == 0) {
297   output("done");
298   output("Generating code...");
299   if (NOGEN) {
300     output("skipped");
301   } else {
302     generateCode();
303     output("done");
304   }
305
306 if (katjaWarningList.size()>0) {
307   showWarnings();
308   output("Warnings:"+katjaWarningList.size());
309   output("\nBUILD\nSUCCESSFUL");
310   showTotalTime(startmillis);
311 } else {
312   output("failed");
313   showBuildFailedAndExit();
314 } else {
315   output("failed");
316   showBuildFailedAndExit();
317 }
318
319 private Analyser analyse() {
320   Analyser analyser = new Analyser(this);
321   analyser.analyse();
322   return analyser;
323 }
324
325 private void generateCode() throws IOException {
326   generateImportsCode();
327   PackageGenerator pkgGen = new PackageGenerator(this, destDir);
328   pkgGen.generate();
329 }
330
331 /**
332 * Generates the code of all imported
333 * Katja specifications
334 * @throws IOException if a problem during the writing of the files occurs
335 */
336 private void generateImportsCode() throws IOException {
337   Iterator it = katjaImports.iterator();
338   while(it.hasNext()) {
339     ((Katja)it.next()).generateCode();
340   }
341 }
342
343 /**
344 * Only parses the specification file, but doesn’t
345 * analyse or generate something
346 * @return the KatjaParser instance of the parser
347 * @throws FileNotFoundException if the
348 * @throws Exception
private Parser parseFile() throws FileNotFoundException, Exception {
    Scanner lexer = new Scanner(new FileReader(specFile));
    lexer.setFilename(specFile);
    Parser parser = new Parser(lexer);
    parser.setKatja(this);
    parser.parse();
    if (parser.numberOfErrors() == 0) {
        spec = parser.getSpecification();
        parseImports();
    }
    return parser;
}

private void checkFileExistence() {
    if (! (new File(destDir)).exists()) {
        error("The destination directory "+destDir+" doesn't exist!\n");
        showBuildFailedAndExit();
    }
    if (! (new File(destDir)).canWrite()) {
        error("You have no write permissions for the destination directory "
              +destDir+"!\n");
        showBuildFailedAndExit();
    }
    if (! (new File(specFile)).exists()) {
        error("The specified file "+specFile+" doesn't exist!\n");
        showBuildFailedAndExit();
    }
}

/**
 * Creates a new KatjaError and adds it to katjaErrorList
 * @param errormsg the message of the error
 * @param file the file where the error have been occurred
 * @param line the line of the error
 */
public void error(String errormsg, String file, KatjaInt line) {
    addError(new KatjaError(new KatjaString(errormsg),
                            new KatjaString(file), line));
}

/**
 * Creates a new KatjaError with the given message
 * and adds it to katjaErrorList
 * @param errormsg the error message
 */
public void error(String errormsg) {
    addError(new KatjaError(new KatjaString(errormsg),
                            new KatjaString(""), new KatjaInt(-1)));
}

/**
 * Prints all error messages stored
 * in katjaErrorList
 */
private void showErrors() {
    KatjaErrorMessage it = katjaErrorMessage.iterator();
    while (it.hasNext()) {
        KatjaErrorMessage e = it.next();
        String msg = "Error";
        if (!e.file().toString().equals(""))
            msg += "in file" + e.file();
        if (!e.line().eq(new KatjaInt(-1))) {
            msg += ":" + e.line();
        }
        msg += ";" + e.message();
        outputln(msg);
    }
}

/**
 * Prints all warn messages stored
 * in katjaWarningList
 */
private void showWarnings() {
    KatjaErrorMessage it = katjaErrorMessage.iterator();
    while (it.hasNext()) {
        KatjaErrorMessage e = it.next();
        String msg = "Warning";
        if (!e.file().toString().equals(""))
            msg += "in file" + e.file();
        if (!e.line().eq(new KatjaInt(-1))) {
            msg += ":" + e.line();
        }
        msg += ";" + e.message();
        outputln(msg);
    }
}

/**
 * Shows all warnings and errors and exits with
 * result 1
 */
private void showBuildFailedAndExit() {
    showWarnings();
    showErrors();
    outputln("\nWarnings: "+katjaErrorMessage.size());
    outputln("Errors: "+katjaErrorMessage.size() + 
"n");
    outputln("BUILD_FAILED");
    showTotalTime(startmillis);
    System.exit(1);
}

/**
 * Parses all imported Katja files
 */
private void parseImports() throws FileNotFoundException, Exception {
    ImportList impList = spec.importList();
    ImportListIterator it = impList.iterator();
    katjaImports = new LinkedList();
    while (it.hasNext()) {
        Import imp = it.next();
        if (imp.is(KatjaImport.sort)) {
        }
    }
}

String importFile = ((KatjaImport) imp).name().toString();

debug("Importing file "+importFile);
File file = new File(specFile);
File impFile = new File(file.getParent(), importFile);
if (fileAlreadyParsed(impFile.getPath())) {
    error("Cyclic import of file "+importFile+"!", specFile, imp.
            line());
} else {
    Katja m = new Katja(impFile.getPath(), this);
    m.parseFile();
    katjaImports.add(m);
    this.nextsortint = m.nextsortint;
}

/**
 * Returns whether the given Katja file gets
 * parsed in a parent Katja file already
 * @param filePath the file to check
 * @return whether the given Katja file gets
 * parsed in a parent Katja file already
 */
private boolean fileAlreadyParsed(String filePath) {
    if (specFile.equals(filePath))
        return true;
    if (parentKatja != null) {
        return parentKatja.fileAlreadyParsed(filePath);
    }
    return false;
}

/**
 * Does a <code>System.out.println</code> if
 * not in quiet mode
 * @param string the string to print out
 */
public static void outputln(String string) {
    if (QUIET)
        return;
    System.out.println(string);
}

/**
 * Does a <code>System.out.print</code> if
 * not in quiet mode
 * @param string the string to print out
 */
public static void output(String string) {
    if (QUIET)
        return;
    System.out.print(string);
}

public String getSpecFile() {

}
public SortIdList getVariantsOfSortId(SortId sortId) {
    return (SortIdList) variantHash.get(sortId.name());
}

/**
 * Returns a <code>SortIdList</code> containing all
 * <code>SortId</code>'s of variants where sortId
 * is contained in.
 * @param sortId the <code>SortId</code> to get the list of.
 * @return a <code>SortIdList</code> containing all
 * <code>SortId</code>'s of variants where sortId
 * is contained in.
 */
public SortIdList getVariantsOfSortId(SortId sortId) {
    return (SortIdList) variantHash.get(sortId.name());
}

/**
 * Fills the variantHash field
 */
private void fillVariantHashtable() {
    ProductionListIterator it = spec.prodList().iterator();
    while (it.hasNext()) {
        Production prod = it.next();
        if (prod.is(VariantProd.sort)) {
            addVariantToHashtable((VariantProd) prod);
        }
    }
    fillVariantHashtableOfKatjaImports();
}

private void fillVariantHashtableOfKatjaImports() {
    Iterator katjalt = getKatjaImports();
    while (katjalt.hasNext()) {
        Katja m = (Katja) katjalt.next();
        m.fillVariantHashtable();
    }
}

/**
 * Adds a VariantPod to the variantHash Hashtable.
 *<br>
 * This means for every SortId contained in prod it adds
 * prod to the SortIdList of the Hashtable with the key
 * of the SortId
 * @param prod the VariantPod to add to the variantHash Hashtable
 */
private void addVariantToHashtable(VariantPod prod) {
    SortIdListIterator it = prod.params().iterator();
    while (it.hasNext()) {
        SortId sortId = it.next();
        SortIdList variantList = (SortIdList) variantHash.get(sortId.name());
        int hashCode = sortId.hashCode();
        if (variantList == null) {
            variantList = new SortIdList();
        }
        variantHash.put(sortId.name(), variantList.appBack(prod.sortId()));
        debug("Adding " + prod.sortId().name() + " to " + sortId.name() + ":hash: "+ hashCode);
    }
}
/**
 * Prints the copyright note to the standard output
 * if the QUIET flag is not set
 */
private static void showCopyright() {
    if (QUIET)
        return;
    System.out.println("---------");
    System.out.println("(c) copyright 2003 AG Software Technology");
    System.out.println("Universität Kaiserslautern, Germany");
    System.out.println();
}

/**
 * Prints the total amount of time needed
 * to the standard output if the QUIET flag
 * is not set.
 * @param startMillis the milliseconds of the start time
 */
private void showTotalTime(long startMillis) {
    if (QUIET)
        return;
    long endMillis = System.currentTimeMillis();
    long diffMillis = endMillis - startMillis;
    long hours = (diffMillis / 3600000);  
    long minutes = (diffMillis / 60000) % 60;
    long seconds = (diffMillis / 1000) % 60;
    long millis = diffMillis % 1000;
    System.out.print("Total time: ");
    if (hours > 0)
        System.out.print(hours + " hours");
    if (minutes > 0)
        System.out.print(minutes + " minutes");
    if (seconds > 0)
        System.out.print(seconds + " seconds");
    if (millis > 0)
        System.out.print(millis + " ms");
    System.out.println();
    System.out.println();
}

/**
 * Prints the usage to the standard output and exit(1) after that
 * if the QUIET flag is not set
 */
private static void showUsageAndExit() {
    if (! QUIET) {
        System.out.println("Usage: javakatja.katja[options]<filename>");
        System.out.println();
        System.out.println("Options:");
        System.out.println("-h|--help show this help");
        System.out.println("-d|--dir=show this help");
    }
}
System.out.println("-dest=<dir> generate the Java classes to <dir>");
System.out.println("If this is not set, the classes are generated to the current working directory.");
System.out.println("-quiet=-quiet be quiet, suppress any output");
System.out.println("-nogen=-nogen only analyse, but don’t generate anything");
System.out.println("-sortint=-sortint an analyser, but don’t generate anything");
System.out.println();
System.exit(1);

/**
 * @param args the arguments
 */
public static void main(String[] args) throws Exception {
    String destDir = null;
    String specFile = null;
    if (args.length < 1) {
        System.out.println("No filename specified!
");
        showUsageAndExit();
    }
    int startSortInt = 0;
    for (int i = 0; i < args.length; i++) {
        if (args[i].equals("-q") ||
            args[i].equals("-quiet"))
            Katja.QUIET = true;
        else if (args[i].equals("-debug"))
            Katja.DEBUG = true;
        else if (args[i].equals("-nogen"))
            Katja.NOGEN = true;
        else if (args[i].equals("-help") ||
            args[i].equals("-h"))
            showUsageAndExit();
        else if (args[i].startsWith("-dest=") ||
            args[i].startsWith("-dest="))
            destDir = getParamValue(args[i]);
        else if (args[i].startsWith("-sortint="))
            startSortInt = Integer.parseInt(getParamValue(args[i]));
        else if (args[i].startsWith("-")) {
            outputln("Unknown option: +" + args[i] + 
"n");
            showUsageAndExit();
        }
        specFile = args[ args.length - 1];
        if (specFile.startsWith("-")) {
            outputln("\nNo filename specified!
");
showUsageAndExit();

Katja katja = new Katja(specFile, destDir);

if (startSortInt > 0) {
    if (startSortInt < 100) {
        outputln("The starting sort integer has to be greater or equal to 100");
        showUsageAndExit();
    } else
        katja.nextsortint = startSortInt;
}

katja.parse();

/**
 * Returns the value of an option.<br>
 * E.g. for the option '-dest=/tmp' it would return '/tmp'
 * @param string an option
 * @return value of an option
 */
private static String getParamValue(String string) {
    return string.substring(string.indexOf("=") + 1, string.length());
}

public static void debug(String s) {
    if (DEBUG)
        System.out.println(s);
}

public Specification getSpecification() {
    return spec;
}

/**
 * Checks whether the given String s is a built-in name
 * such as String or Int and returns the Java name for
 * it like KatjaString or KatjaInt.
 * @param s the String to check
 * @return a String that can be used
 */
public static String getBuiltInName(String s) {
    String s2 = (String) builtInNames.get(s);
    if (s2 == null)
        return s;
    return s2;
}

/**
 * Returns whether the given String s is a reserved word in Java
 * @param s the String to check
 * @return whether the given String s is a reserved word in Java
 */
public static boolean isReservedWord(String s) { return reservedWords.contains(s); }
/**
 * Returns whether the given String s is a Java native type
 * @param s the String to check
 * @return whether the given String s is a Java native type
 */
public static boolean isNativeType(String s) {
    return nativeTypes.contains(s);
}

/**
 * Returns whether the given String s is a Katja built-in type
 * @param s the String to check
 * @return whether the given String s is a Katja built-in type
 */
public static boolean isBuiltInType(String s) {
    return builtInNames.containsValue(s);
}

/**
 * Returns a free sort integer
 * @return a free sort integer
 */
public int getFreeSortInt() {
    return nextSortInt++;
}

/**
 * Returns an Iterator of Katja objects of all imported Katja files
 * @return an Iterator of Katja objects of all imported Katja files
 */
public Iterator getKatjaImports() {
    return katjaImports.iterator();
}

/**
 * Returns the full qualified name of the given name
 * @param name the Java type of which the full qualified name should be returned
 * @return the full qualified name of the given name
 */
public JavaImport getJavaImportForName(String name) {
    return (JavaImport) javaImportHash.get(name);
}

/**
 * Returns the package name of the given sort name
 * @param sortName the sort name to get the package of
 * @return the package name of the given sort name
 */
public String getPackageOfSortId(String sortName) {
    return (String) sortIdPackageHash.get(sortName);
}

/**
 * Adds the given error to the internal error list
 * @param error the error to add to the internal error list
 */
public void addError(KatjaError error) {
    if (parentKatja == null)
        katjaErrorList = katjaErrorList.add(error);
}
else
    parentKatja.addError(error);
}

/**
 * Adds the given warning to the internal warning list
 * @param warning the warning to add to the internal warning list
 */
public void addWarning(KatjaWarning warning) {
    if (parentKatja == null)
        katjaWarningList = katjaWarningList.add(warning);
    else
        parentKatja.addWarning(warning);
}

package katja;
import java.io.File;
import java.io.IOException;
import katja.spec.ListProd;
import katja.spec.PackageDcl;
import katja.spec.PackageName;
import katja.spec.ProductionList;
import katja.spec.TupleProd;
import katja.spec.VariantProd;

/**
 * The MaxPackageGenerator is responsible for creating
 * a package containing generated classes.
 * You initialize it with the package name and the
 * destination directory where to generate the classes.
 * After that you call the different methods to create the
 * desired classes.
 * @author Jan Schäfer
 */
public class PackageGenerator {

    Katja katja;

    /**
     * This variable holds the name of the
     * package for all created classes
     */
    private String packageName = "";

    /**
     * This variable holds the directory where
```java
* the generated classes should be put in.
*/
private File destDir;

/**
 * This constructor creates a new <code>MaxClassGenerator</code> object
 * an sets the package name and the destination directory
 * where the generated classes should be put in.
 * Notice that the generated classes will be put under
 * the destination directory plus the package directory.
 * E.g. if destDir is &lt;i&gt;/tmp/test&lt;/i&gt; and the package name
 * is &lt;i&gt;my.katja.test&lt;/i&gt; the classes would be put into the
 * directory &lt;i&gt;/tmp/test/my/katja/test&lt;/i&gt;
 * @param katja the Katja instance
 * @param destDir the destination directory of the generated classes
 */
public PackageGenerator(Katja katja, String destDir) throws IOException {
    this.katja = katja;
    PackageDcl packageDcl = katja.getSpecification().pck();
    if (packageDcl.is(PackageName.sort))
        this.packageName = ((PackageName) packageDcl).name().stringValue();
    this.destDir = new File(destDir, getPackageDirectory());
    if (!this.destDir.exists()) {
        this.destDir.mkdirs();
    }
}

/**
 * Returns the directory of the package name.
 * E.g. if the package name is &lt;i&gt;my.katja.test&lt;/i&gt; the
 * result would be &lt;i&gt;my/katja/test&lt;/i&gt; on a UNIX system
 * resp. &lt;i&gt;my\katja\test&lt;/i&gt; on a Windows(R) system
 * @return the directory for the package name
 */
protected String getPackageDirectory() {
    return packageName.replace('.', File.separatorChar);
}

/**
 * Return the directory where classes will be generated to,
 * including the package directory
 * @return the directory where classes will be generated to.
 */
public File getDestinationDir() {
    return destDir;
}

/**
 * Returns the package name of the generated classes
 * @return the package name of the generated classes
 */
public String getPackageName() {
    return packageName;
}

public void generate() throws IOException {
    ProductionList prodList = katja.getSpecification().prodList();
    for (int i = 0; i < prodList.numSubterms(); i++) {
```
```
private void generate(Production prod) throws IOException {
    switch (prod.sort().toInt()) {
        case TupleProd.sortInt : generate((TupleProd) prod); break;
        case ListProd.sortInt : generate((ListProd) prod); break;
        case VariantProd.sortInt : generate((VariantProd) prod); break;
    }
}

private void generate(TupleProd prod) throws IOException {
(new TupleGenerator(this, prod)).generate();
}

private void generate(ListProd prod) throws IOException {
(new ListGenerator(this, prod)).generate();
}

private void generate(VariantProd prod) throws IOException {
(new VariantGenerator(this, prod)).generate();
}
```

---

```
package katja;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintStream;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.ListIterator;
import java.util.Set;
import java.util.Collection;
import katja.spec.JavaImport;
import katja.spec.Production;
import katja.spec.SortIdList;
import katja.spec.SortIdListIterator;

/**
 * This class is the super class for the specialized generators. It implements common methods
 * to generate Java classes.
 * @author Jan Schafwer
 */
```
```java
public abstract class ClassGenerator {

    /**
     * Holds destination directory and package name
     */
    protected PackageGenerator pkgGen;

    /**
     * Holds the name of the class to be generated
     */
    protected String className;

    /**
     * The Production to create the class from
     */
    protected Production prod;

    /**
     * The Writer to write the generated class
     */
    protected PrintWriter writer;

    /**
     * Creates a new ClassGenerator
     * @param packageGenerator the PackageGenerator
     * @param prod the Production to create the class from
     * @author Jan Schäfer
     */
    public ClassGenerator(
            PackageGenerator packageGenerator,
            Production prod)
    {
        this.pkgGen = packageGenerator;
        this.className = prod.sortId().name().stringValue();
        this.prod = prod;
    }

    /**
     * Creates a new ClassGenerator
     * @param packageGenerator
     * @param className the name of the class to be generated
     */
    public ClassGenerator(
            PackageGenerator packageGenerator,
            String className)
    {
        this.pkgGen = packageGenerator;
        this.className = className;
    }

    /**
     * Generates the Java file
     * @throws IOException if a problem during the writing of the file occurs
     */
    public void generate() throws IOException {
        this.writer = new PrintWriter(
                new BufferedWriter(
                        new FileWriter(getFilePath())));
        generatePackageName();
        generateImportList();
        generateClassDeclaration();
        generateClassBody();
        this.writer.close();
    }
}
```
private void generatePackageDeclaration() {
    if (this.pkgGen.getPackageName().equals(""))
        return;

    writer.println("package "+this.pkgGen.getPackageName()+";");
    writer.println(); // Empty line after the package declaration
}

/**
 * Generates the list of import declarations.
 * It uses for this the List that is returned by the getImportList method.
 */
private void generateImportList() {
    Iterator it = getImportList().iterator();

    while (it.hasNext()) {
        writer.println("import "+it.next()+";");
    }

    writer.println(); // Empty line after the import list
}

private void generateClassDeclaration() {
    writer.print("public ");

    if (isInterface())
        writer.print("interface ");
    else
        writer.print("class ");

    writer.println(className);
    if (getExtendedClass() != null)
        writer.println("extends "+getExtendedClass()+" ");
    generatedImplementedInterfaces();
    writer.println();
}

/**
 * Generates the list of implemented interfaces.
 * It uses for this the List returned by the getImplementedInterfaces method.
 */
private void generatedImplementedInterfaces() {
    ListIterator it = getImplementedInterfaces().listIterator();

    if (it.hasNext()) {
        // if its an interface we have to extend the other interfaces
        // instead of implementing them
        if (isInterface())
            writer.print(",");
        else
            writer.print("implements ");
    }

    while (it.hasNext()) {

writer.print(it.next());
if (it.hasNext())
    writer.print(“,“);
}

/**
 * Generates the body of the class.
 */
private void generateClassBody() {
    writer.println("{");
    generateSortDefinition();
    generateMemberVariables();
    generateMethods();
    generateSortMethod();
    writer.println("}"
    );
}

/**
 * Generates the KatjaSort related static members:
 * <pre>
 * public static int sortInt = ...
 * public static KatjaSort sort = ...
 * </pre>
 */
protected void generateSortDefinition() {
    // For interfaces there is no static sortInt defined, because it
    // is never needed.
    if (!isInterface())
        writer.println("public static final int sortInt = “+prod.sortInt
        ();
    writer.println("public static final KatjaSort sort = new KatjaSort("+
    className+".class,”+prod.sortInt()+”);
    writer.println();
}

/**
 * Generates all member variables.<br>
 * Has to be overridden by specialized class
 * generators.
 */
protected abstract void generateMemberVariables();

/**
 * Generates all methods.<br>
 * Has to be overridden by specialized class
 * generators.
 */
protected abstract void generateMethods();

/**
 * Generates the sort() method
 */
protected void generateSortMethod() {
    writer.print("public KatjaSort sort() “);
    if (isInterface())
        writer.println(“;”);
    else {
        writer.println("{“);
        writer.println("return new KatjaSort("+
        className+".sort; “);
    writer.println("}");
}
```java
Returns the filename of this class
@return the filename of this class
*/
public String getFileName() {
    return className + " .java";
}

/**
 * Returns the path of the file of this class
 * @return the path of the file of this class
 */
public File getFilePath() {
    return new File(this.pkgGen.getPackageDirectory(), getFileName());
}

/**
 * Returns the class name that should be extended by this
 * class or null if the class doesn't extend
 * any class.
 * This method must be overridden by specialized
 * generator classes.
 * @return the class that should be extended by this class.
 */
protected abstract String getExtendedClass();

/**
 * Returns a list of String objects, that contains the
 * interface names of interfaces which should be implemented
 * by this class. This method must be overridden by specialized
 * generator classes. <br>
 * When override this method don't forget to call the super method
 * to not lost the implemented interfaces of the super class
 * @return
 */
protected List getImplementedInterfaces() {
    List interfaces = new LinkedList();
    if (prod != null) {
        SortIdList variantList = pkgGen.katja.getVariantsOfSortId(prod.sortId);
        if (variantList == null)
            return interfaces;
        SortIdListIterator it = variantList.iterator();
        while (it.hasNext()) {
            interfaces.add(it.next().name().toString());
        }
    }
    return interfaces;
}

/**
 * Returns a list of String objects, that contains the
 * full qualified imported classes. This method should
 * be overridden in specialized generator classes.
 */
```
Notice to call the super() method to not lose the imported classes of this class. Which is in fact <code>katja.common.*</code><br>

@return a list of String objects, that contains the full qualified imported classes

protected Collection getImportList() {
    Set result = new LinkedHashSet();
    result.addAll(getImportListOfSortIdNames());
    result.addAll(getImportListOfImplementedInterfaces());
    return result;
}

private List getImportListOfImplementedInterfaces() {
    Iterator it = implementedInterfaces().iterator();
    LinkedList result = new LinkedList();
    while (it.hasNext()) {
        String name = (String) it.next();
        if (name != null) {
            String qualifiedName = getFullQualifiedName(name);
            if (qualifiedName != null)
                result.add(qualifiedName);
        }
    }
    return result;
}

/**
 * Returns a List of String objects of full qualified class names which are generated of the List returned by
 * the getSortIdName method
 * @return a List of String objects of full qualified class names which are generated of the List returned by
 * the getSortIdName method
 */

private List getImportListOfSortIdNames() {
    List result = new LinkedList();
    Iterator it = getSortIdNames().iterator();
    while (it.hasNext()) {
        String sortIdName = (String) it.next();
        String fullQualifiedName = getFullQualifiedName(sortIdName);
        if (fullQualifiedName != null &&
            !result.contains(fullQualifiedName))
            result.add(fullQualifiedName);
    }
    return result;
}

/**
 * Returns the full qualified name of a class name<br>
 * There are two possibilities to find out the full qualified name. Either the name is an import Java type, then the Java
 * import statement is used, or the type is a Katja type then the package name plus the type is returned.<br>
 * If nothing works and the full qualified name would be the same as the given parameter, <code>null</code> is returned
 */
* E.g. for 'List' it would return 'java.util.List' iff
* there is an 'import java.util.List' statement in the Katja
* file.<br>
* for 'Exp' it would return 'mypackage.Exp' if there is an
* 'package mypackage' statement in the Katja file where Exp is
* defined.
*<br>
* @param sortIdName the name to find out the full qualified name
* @return The full qualified name of the given name, or <code>null</code>
* if the return name would be the same as the parameter
*/
private String getFullQualified(String sortIdName)
{
    // Test if it is a Java type
    JavaImport imp = pkgGen.katja.getJavaImportForName(sortIdName);
    if (imp != null) {
        return imp.name().toString();
    } else { // If no Java type look for the packagename
        String pck = pkgGen.katja.getPackageOfSortId(sortIdName);
        if (pck != null)
            Katja.debug(sortIdName+"sort has package: "+pck);
        else
            Katja.debug(sortIdName+"sort has no package");

        // If a package name is declared return the package name
        // plus the sort name, otherwise return null
        if (pck != null & & !pck.equals("") & & !pck.equals(pkgGen.getPackageName())) {
            return pck+"."+sortIdName;
        } else
            return null;
    }
}

/**
* Returns a List of String objects of the
* names of sorts that are used in the subclass.<br>
* This List is then used to create the needed imports.
* @return a List of String objects of the
* names of sorts that are used in the subclass.
*/
protected abstract List getSortIdNames();

/**
* Returns whether an interface rather a class should be generated<br>
* The default implementation returns false.<br>
* Specialized generators can override this to generate an interface
* rather a class
* @return whether an interface rather a class should be generated
*/
protected boolean isInterface() {
    return false;
}
package katja;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import katja.common.KatjaString;
import katja.spec.Selector;
import katja.spec.SortId;
import katja.spec.TupleParam;
import katja.spec.TupleParamListIterator;
import katja.spec.TupleProd;

/**
 * This class generates a class from a Katja tuple.
 * @author Jan Schaffer
 */
public class TupleGenerator extends ClassGenerator {

    private TupleProd tupleProd;

    /**
     * Creates a new KatjaTupleGenerator
     * @param packageGenerator the PackageGenerator
     * @param prod the TupleProd to generate the class from
     */
    public TupleGenerator(PackageGenerator packageGenerator,
                          TupleProd prod)
    {
        super(packageGenerator, prod);
        this.tupleProd = prod;
    }

    /**
     * Generates all member variables.
     * Which are one variable for every child
     * @see katja.ClassGenerator#generateMemberVariables()
     */
    protected void generateMemberVariables() {
        if (tupleProd.params().numSubterms() == 0)
            return;
        for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
            TupleParam param = tupleProd.params().subterm(i);
            SortId sortId = getSortIdFromTupleParam(param);
            writer.println("private " + sortId.subterm(0) + " child" + i + ":" );
        }
        writer.println();
    }

    private SortId getSortIdFromTupleParam(TupleParam param) {
        SortId sortId;
        if (param.is(Selector.sort))
            sortId = ((Selector)param).sortId();
        else
            sortId = (SortId) param;
        return sortId;
    }
}
protected void generateMethods () {
    generateConstructor ( ) ;
    generateNumSubterms ( ) ;
    generateGet ( ) ;
    generateSubterm ( ) ;
    generateEq ( ) ;
    generateSelectorFunctions ( ) ;
    generateHashCode ( ) ;
    generateToString ( ) ;
}

private void generateConstructor ( ) {
    // We do not need an empty constructor
    if ( tupleProd . params ( ) . numSubterms ( ) == 0 )
        return ;
    writer . print ( "public " + className + " ( " ) ;
    generateConstructorArguments ( ) ;
    writer . println ( " ) ;
    generateConstructorBody ( ) ;
    writer . println ( " ; " ) ;
}

private void generateConstructorArguments ( ) {
    for ( int i = 0 ; i < tupleProd . params ( ) . numSubterms ( ) ; i ++ ) {
        TupleParam param = tupleProd . params ( ) . subterm ( i ) ;
        SortId sortId = getSortIdFromTupleParam ( param ) ;
        writer . print ( sortId . name ( ) + " arg" + i ) ;
        if ( i < tupleProd . params ( ) . numSubterms ( ) - 1 )
            writer . print ( " , " ) ;
    }
}

private void generateConstructorBody ( ) {
    for ( int i = 0 ; i < tupleProd . params ( ) . numSubterms ( ) ; i ++ ) {
        writer . println ( "this . child" + i + " = arg" + i + " ; " ) ;
    }
}

private void generateNumSubterms ( ) {
    writer . println ( "public int numSubterms ( ) { " ) ;
    writer . println ( "return " + tupleProd . params ( ) . numSubterms ( ) + " ; " ) ;
    writer . println ( " } ; " ) ;
}

private void generateGet ( ) {
    writer . println ( "public Object get ( int ith ) { " ) ;
    writer . println ( "return subterm ( ith ) ; " ) ;
    writer . println ( " } ; " ) ;
}

/**
 * Generates the subterm method.</br>
 * If the class has no subterms at all, this
 * method isn't generated.
 */
private void generateSubterm() {
    // Don't generate the subterm method if the
tuple class has no subterms
    if (prod.params().numSubterms() == 0)
        return;
    writer.println("public Object subterm(int ith) {");
    writer.println("switch(ith) {");
    for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
        writer.println("case " + ith + " : return child " + ith + ";");
    }
    writer.println("default : ");
    writer.println("if (ith < 0) throw new KatjaIllegalArgumentOutOfRangeException(");
    writer.println("" + tupleProd.params().numSubterms() + "" + ", but sort " + sort().toString() + " has only " + tupleProd.params().numSubterms() + " subterms!");
    writer.println(");
    writer.println("return false;");
    writer.println(");
    if (tupleProd.params().numSubterms() > 0) {
        writer.println("return true;");
        writer.println(");
        for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
            writer.println("if (!t.child " + ith + ".equals(" + ith + ")");
            writer.println("return false;");
        }
    }
    writer.println(");
}

private void generateEq() {
    writer.println("public boolean eq(KatjaElement e) {");
    writer.println("if (!e.is(" + className + ".sort)) return false;

    writer.println(");
    writer.println("if (tupleProd.params().numSubterms() > 0) {
        writer.println("return true;");
        writer.println(");
        for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
            writer.println("generateSelectorFunction((Selector) param, i);
        }
    }
}

private void generateSelectorFunctions() {
    for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
        writer.println("TupleParam param = tupleProd.params().subterm(i);
        if (param.is(Selector.sort)) {
            generateSelectorFunction((Selector) param, i);
        }
    }
}
private void generateSelectorFunction(Selector selector, int ithArg) {
    SortId sortId = selector.sortId();
    KatjaString selName = selector.selector();
    writer.print("/*public*/");
    writer.println("sortId.name()=" + sortId.name());
    writer.println("selName+" + selName);
    writer.println("return child+ithArg+" + ithArg);
    writer.println(";");
}

/**
   * Generates the hashCode function
   */
private void generateHashCode() {
    writer.println("public int hashCode() {");
    writer.println("return +tupleProd.sortInt();");
    for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
        writer.println("+child +" + i + " . hashCode()*31");
    }
    writer.println(";");
    writer.println("}
    
/**
   * Generates the toString function
   */
private void generateToString() {
    writer.println("public StringBuffer toString() {");
    writer.println("result = new StringBuffer();");
    writer.println("for (int i = 0; i < tupleProd.params().numSubterms(); i++) {
        writer.println("result . append(" + tupleProd.params().numSubterms());
        if (i + 1 < tupleProd.params().numSubterms())
            writer.println("result . append(" + tupleProd.params().numSubterms());
    }
    writer.println("result . append(" + tupleProd.params().numSubterms());
    writer.println("result . toString();
    writer.println();
}

/**
   * Returns 'KatjaElementImpl'
   */
* @see katja.ClassGenerator#getExtendedClass()
protected String getExtendedClass() {
    return "KatjaTupleImpl";
}

protected List getSortIdNames() {
    List list = new LinkedList();
}
```
TupleParamListIterator it = tupleProd.params().iterator();
while (it.hasNext()) {
    TupleParam param = it.next();
    SortId sortId;
    if (param.is(SortId.sort))
        sortId = (SortId) param;
    else
        sortId = ((Selector) param).sortId();
    list.add(sortId.name().toString());
}
return list;
}

protected Collection getImportList() {
    Collection col = super.getImportList();
    col.add(new String("katja.common.*"));
    return col;
}

/*
 * Copyright 2003
 * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
 * All rights reserved.
 */
package katja;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import katja.common.KatjaInt;
import katja.common.KatjaString;
import katja.spec.KatjaWarning;
import katja.spec.Production;
import katja.spec.ProductionList;
import katja.spec.ProductionListIterator;
import katja.spec.Selector;
import katja.spec.SelectorList;
import katja.spec.SelectorListIterator;
import katja.spec.SortId;
import katja.spec.SortIdList;
import katja.spec.SortIdListIterator;
import katja.spec.TupleParam;
import katja.spec.TupleParamListIterator;
import katja.spec.TupleProd;
import katja.spec.VariantProd;

/**
 * Create a class from a VariantProd
 * @author Jan Schaffer
 */
public class VariantGenerator extends ClassGenerator {
    private SortIdList children;
```
private VariantProd variantProd;
private SelectList allSelectors = null; // Lazy initialisation

/**
 * Creates a new KatjaVariantGenerator
 * @param packageGenerator the PackageGenerator
 * @param prod the VariantProd to create the class from
 */
public VariantGenerator(
    PackageGenerator packageGenerator,
    VariantProd prod)
{
    super(packageGenerator, prod);
    this.children = prod.params();
    this.variantProd = prod;
}

/**
 * Does nothing, as interfaces have no member variables
 * @see katja.ClassGenerator#generateMemberVariables()
 */
protected void generateMemberVariables() {
}

/**
 * Generates all SelectorFunctions that are the same in all subtypes
 * @see katja.ClassGenerator#generateMethods()
 */
protected void generateMethods() {
    generateSelectorFunctions();
}

private void generateSelectorFunctions() {
    SelectListIterator it = getAllSelectors().iterator();
    while(it.hasNext()) {
        generateSelectorFunction(it.next());
    }
}

private void generateSelectorFunction(Selector selector) {
    SortId sortId = selector.sortId();
    KatjaString selName = selector.selector();
    writer.print("\_\_\_public \_");
    writer.print(sortId.name() + "+");
    writer.println(selName+"()");
    writer.println();
}

private SelectList getAllSelectors() {
    if (allSelectors == null)
        allSelectors = getAllSelectors(variantProd);
    return allSelectors;
}

private SelectList getAllSelectors(VariantProd variant) {
    ProductionListIterator it = getAllProductions(variant).iterator();
    SelectList list = new SelectList();
    // First add all Selectors of the first TupleProduction
if (it.hasNext()) {
  Production p = it.next();
  if (p.is(TupleProd.sort)) {
    list = getAllSelectors((TupleProd) p);
  } else if (p.is(VariantProd.sort)) {
    list = getAllSelectors((VariantProd) p);
  } else {
    // If there is a ListProduction there are no Selectors
    return new SelectorList();
  }
}

// Then merge the others
while (it.hasNext()) {
  Production p = it.next();
  if (p.is(TupleProd.sort)) {
    list = mergeSelectorLists(list, getAllSelectors((TupleProd) p));
  } else if (p.is(VariantProd.sort)) {
    list = mergeSelectorLists(list, getAllSelectors((VariantProd) p));
  } else {
    // If there is a ListProduction there are no Selectors
    return new SelectorList();
  }
}
return list;

private ProductionList getAllProductions(VariantProd variant) {
  ProductionListIterator it = pkgGen.katja.getSpecification().prodList().iterator();
  ProductionList list = new ProductionList();
  while (it.hasNext()) {
    Production p = it.next();
    if (variantContainsSortId(variant, p.sortId())) {
      list = list.add(p);
    }
  }
  return list;
}

private boolean variantContainsSortId(VariantProd variant, SortId id) {
  SortIdListIterator it = variant.params().iterator();
  while (it.hasNext()) {
    SortId sortId = it.next();
    if (sortId.name().eq(id.name()))
      return true;
  }
  return false;
}

private SelectorList getAllSelectors(TupleProd p) {
  TupleParamListIterator it = p.params().iterator();
  SelectorList result = new SelectorList();
  while (it.hasNext()) {
    TupleParam param = it.next();
    if (param.is(Selector.sort)) {
      result = result.add((Selector) param);
    }
  }
  return result;
}
private SelectorList mergeSelectorLists(SelectorList a, SelectorList b) {
    SelectorListIterator it = a.iterator();
    SelectorList result = new SelectorList();
    while (it.hasNext()) {
        Selector sel = it.next();
        if (selectorListContains(b, sel))
            result = result.add(sel);
        else {
            Selector sel2 = getSelectorWithName(b, sel.selector());
            if (sel2 != null) {
                pkgGen.katja.addWarning(new KatjaWarning(new KatjaString("Conflicting types of selectors!
" + "Can't add selector " + sel.selector() + " to variant " +
"+classname+".'", new KatjaString(""), new KatjaInt(-1)));
                continue;
            }
        }
    }
    return result;
}

private boolean selectorListContains(SelectorList b, Selector s) {
    SelectorListIterator it = b.iterator();
    while (it.hasNext()) {
        Selector sel2 = it.next();
        if (sel2.selector().eq(sel.selector()))
            if (sel2.sortId().name().eq(sel.sortId().name()))
                return true;
    }
    return false;
}

private Selector getSelectorWithName(SelectorList list, KatjaString string) {
    SelectorListIterator it = list.iterator();
    while (it.hasNext()) {
        Selector sel = it.next();
        if (sel.selector().eq(string))
            return sel;
    }
    return null;
}

/**
 * Returns 'KatjaElement'
 * @see katja.ClassGenerator#getExtendedClass()
 */
protected String getExtendedClass() {
    return "KatjaElement";
}

protected boolean isInterface() {
    return true;
}

protected List getSortIdNames() {
    LinkedList list = new LinkedList();
// First add all children
SortIdListIterator it = children.iterator();
while (it.hasNext()) {
    list.add(it.next().name().toString());
}

// Then add all return types of selector methods
SelectorListIterator it2 = getAllSelectors().iterator();
while (it2.hasNext()) {
    list.add(it2.next().sortId().name().toString());
}
return list;

protected Collection getImportList() {
    Collection col = super.getImportList();
    col.add(new String("katja.common.");
    return col;
}

ListGenerator.java

/*
 * Copyright 2003
 * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
 * All rights reserved.
 */
package katja;
import java.io.IOException;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import katja.spec.ListProd;

/**
 * Class to generate List classes
 * @author Jan Schäfer
 */
public class ListGenerator extends ClassGenerator {
    String childType;
    ListIteratorGenerator iteratorGen;

    /**
     * Creates a new KatjaListGenerator
     * @param pkgGen the PackageGenerator
     * @param prod the ListProd to create the class from
     *
     * @param pkgGen, prod
     */
    public ListGenerator(
        PackageGenerator pkgGen,
        ListProd prod)
    {
        super(pkgGen, prod);
        this.childType = prod.param().name().stringValue();
        iteratorGen = new ListIteratorGenerator(pkgGen, prod);
    }
public void generate() throws IOException {
    super.generate();
    iteratorGen.generate();
}

/**
 * Generates nothing, as the member variables are defined
 * in KatjaListImpl
 * @see katja.classGenerator#generateMemberVariables()
 */
protected void generateMemberVariables() {
}

protected void generateMethods() {
generateSubterm();
generateGet();
generateContains();
generateFirst();
generateLast();
generateAppFront();
generateAppBack();
generateAdd();
generateRemove();
generateRemoveAll();
generateFront();
generateBack();
generateConc();
generateAddAll();
generateIterator();
generateCreateInstance();
}

private void generateGet() {
    writer.println("public " + childType + " get(int ith) {");
    writer.println("return subtermInternal(ith);");
    writer.println();
}

private void generateContains() {
    writer.println("public boolean contains(" + childType + " e) {");
    writer.println("return containsInternal(e);");
    writer.println();
}

private void generateRemove() {
    writer.println("public " + className + " remove(" + childType + " e) {");
    writer.println("return removeInternal(e);");
    writer.println();
}

private void generateRemoveAll() {
    writer.println("public " + className + " removeAll(" + className + " e) {");
    writer.println("return removeAllInternal(e);");
    writer.println();
}
```java
private void generateAdd() {
    writer.println("public String add(String childType) {");
    writer.println("return add(" + className + ") ;");
    writer.println();
}

private void generateAddAll() {
    writer.println("public String addAll(String childType) {");
    writer.println("return addAll(" + className + ") ;");
    writer.println();
}

private void generateSubterm() {
    writer.println("public String subterm(int ith) {");
    writer.println("return subterm(" + className + ") ;");
    writer.println();
}

private void generateFirst() {
    writer.println("public String first() {");
    writer.println("return first(" + className + ") ;");
    writer.println();
}

private void generateLast() {
    writer.println("public String last() {");
    writer.println("return last(" + className + ") ;");
    writer.println();
}

private void generateAppFront() {
    writer.println("public String appFront(String childType) {");
    writer.println("return appFront(" + className + ") ;");
    writer.println();
}

private void generateAppBack() {
    writer.println("public String appBack(String childType) {");
    writer.println("return appBack(" + className + ") ;");
    writer.println();
}

private void generateFront() {
    writer.println("public String front() {");
    writer.println("return front(" + className + ") ;");
    writer.println();
}

private void generateBack() {
    writer.println("public String back() {");
    writer.println("return back(" + className + ") ;");
    writer.println();
}
```

```java
private void generateConc() {
  writer.println("public " + className + "Conc(" + className + " l) {");
  writer.println("return " + className + "ConcInternal(l);");
  writer.println("};
}

private void generateIterator() {
  writer.println("public " + className + "Iterator iterator () {");
  writer.println("return new " + className + "Iterator(values.iterator());");
  writer.println("};
  writer.println();
}

private void generateCreateInstance() {
  writer.println("protected KatjaListImpl createInstance (List<
  listValues>) {");
  writer.println("+className+ l = new +className+ ();");
  writer.println("l . values = listValues ;");
  writer.println("return l ;");
  writer.println();
}

/**
 * Returns 'KatjaListImpl'
 * @see katja.KatjaClassGenerator#getExtendedClass()
 */
protected String getExtendedClass() {
  return "KatjaListImpl";
}

protected Collection getImportList() {
  Collection col = super.getImportList();
  col.add(new String("katja.common."));
  col.add(new String("java.util.List"));
  return col;
}

protected List getSortIdNames() {
  LinkedList list = new LinkedList();
  list.add(childType);
  return list;
}
```

ListIteratorGenerator.java

```
import java.util.List;
import katja.spec.ListProd;

/**
 * Class to generate ListIterator classes
 * @author Jan Schäfer
 */
public class ListIteratorGenerator extends ClassGenerator {

    ListProd listProd;

    public ListIteratorGenerator(
        PackageGenerator katjaPackageGenerator,
        ListProd prod)
    {
        super(katjaPackageGenerator, prod.sortId().name()+"Iterator");
        listProd = prod;
    }

    protected void generateMemberVariables() {
        writer.println("private Iterator iterator;\n    writer.println();
    }

    protected void generateMethods() {
        generateConstructor();
        generateHasNext();
        generateNext();
    }

    private void generateConstructor() {
        writer.println("public "+className+
            "Iterator it()\n        writer.println("\n            writer.println("\n        writer.println();
    }

    private void generateHasNext() {
        writer.println("public boolean hasNext()\n        writer.println("\n        writer.println("\n        writer.println();
    }

    private void generateNext() {
        writer.println("public "+listProd.param().name()+" next()\n        writer.println("\n        writer.println("\n        writer.println();
    }

    /**
     * returns <code>null</code>
     * @return <code>null</code>
     */
    protected String getExtendedClass() {
        return null;
    }

    /**
     * returns <code>null</code>
     * @return <code>null</code>
     */
}
**Does nothing, as ListIterators have no sort**

```java
protected void generateSortDefinition() {
}
```

/**
 * We don't want any sort code here, so
 * we override the method and do nothing
 */

```java
protected void generateSortMethod() {
}
```

```java
protected Collection getImportList() {
    Collection col = super.getImportList();
    col.add("java.util.Iterator");
    return col;
}
```

```java
protected List getSortIdNames() {
    LinkedList list = new LinkedList();
    list.add(listProd.param().name().toString());
    return list;
}
```

---

**Analyser.java**

```java
/*
 * Copyright 2003
 * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
 * All rights reserved.
 */

package katja;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import katja.common.KatjaInt;
import katja.common.KatjaString;
import katja.spec.*;

/**
 * The Analyser
 */

public class Analyser {
    private Katja katja;

    private int num_errors = 0;

    /**
     * Stores all defined SortIds
     * Defined is a sortid after it has
     * been declared as a tuple, list or variant
     */
    private HashSet definedSortIdHash;

    /**
     * Stores for every SortId the
     */
```
package where it is defined

private HashMap sortIdPackages;

/**
 * Contains all Strings of names that are defined by java import statements.
 */
private HashMap definedJavaImportHash;

public Analyser(Katja katja) {
    this.katja = katja;
    definedSortIdHash = new HashSet();
    definedJavaImportHash = katja.javaImportHash;
    sortIdPackages = katja.sortIdPackageHash;
}

public Analyser(Katja katja, Analyser parent) {
    this.katja = katja;
    this.definedJavaImportHash = parent.definedJavaImportHash;
    this.definedSortIdHash = parent.definedSortIdHash;
    this.sortIdPackages = parent.sortIdPackages;
}

public void analyse() {
    analyseImports();
    fillDefinedHashes();
    checkDefinitionOfSortIds();
}

private void checkDefinitionOfSortIds() {
    Specification spec = katja.getSpecification();
    ProductionListIterator it = spec.prodList().iterator();
    while (it.hasNext()) {
        Production prod = it.next();
        switch (prod.sort().toInt()) {
            case TupleProd.sortInt:
                checkTupleDefinition((TupleProd) prod);
                break;
            case ListProd.sortInt:
                checkListDefinition((ListProd) prod);
                break;
            case VariantProd.sortInt:
                checkVariantDefinition((VariantProd) prod);
                checkCyclicFreeness((VariantProd) prod);
                break;
        }
    }
}

/** Checks if the variant is cyclicly defined<br>
 * E.g. A = A would be a cycle, or A = B B = A, or
 * A = B B = C C = A
 */
private void checkCyclicFreeness(VariantProd prod) {
    findSortIdInVariants(prod.sortId(), prod);
}
private void findSortIdInVariants(SortId variant, VariantProd prod) {
    SortIdList list = katja.getVariantsOfSortId(variant);
    if (list == null)
        return;

    SortIdListIterator it = list.iterator();

    while (it.hasNext()) {
        SortId sortId = it.next();
        if (sortId.eq(prod.sortId()))
            error("Cyclic dependency of variant "+s.sortId().name()+" detected!", prod.sortId().line());
        return;
    }

    findSortIdInVariants(sortId, prod);
}

private void checkVariantDefinition(VariantProd prod) {
    SortIdListIterator it = prod.params().iterator();

    while (it.hasNext()) {
        SortId id = it.next();
        if (!sortIdsDefined(id))
            error("Sort "+id.name()+" is undefined!", id.line());
        else
            if (definedJavaImportHash.containsKey(id.name().toString()))
                error("The variant "+s.sortId().name()+" contains the Java type "+id.name()+" which is not allowed", prod.sortId().line());
            else
                if (Katja.isBuiltInType(id.name().toString()))
                    error("The variant "+s.sortId().name()+" contains the built-in type "+id.name()+" which is not allowed", prod.sortId().line());
    }
}

private void checkListDefinition(ListProd prod) {
    if (!sortIdsDefined(prod.param()))
        error("Sort "+s.param().name()+" is undefined!", prod.sortId().line());
}

private void checkTupleDefinition(TupleProd prod) {
    TupleParamListIterator it = prod.params().iterator();

    while (it.hasNext()) {
        TupleParam param = it.next();
        SortId sortId;
        if (param.is(SortId.sort))
            sortId = (SortId) param;
        else
            sortId = ((Selector) param).sortId();

        if (!sortIdsDefined(sortId))
            error("Sort "+sortId.name()+" is undefined!", sortId.line());
    }
}
private boolean sortIdIsDefined(SortId id) {
    if (Katja.isBuiltInType(id.name().toString()))
        return true;
    if (definedJavaImportHash.containsKey(id.name().toString()))
        return true;
    if (definedSortIdHash.contains(id.name()))
        return true;
    if (isInClassPath(id.name().toString()))
        return true;
    return false;
}

private boolean isInClassPath(String string) {
    if (findClass(string))
        return true;
    if (findClass("java.lang."+string))
        return true;
    return false;
}

private boolean findClass(String string) {
    try {
        this.getClass().getClassLoader().loadClass(string);
        Katja.debug("Sort"+string+" is in classpath!");
        return true;
    } catch (ClassNotFoundException e) {
        Katja.debug("Sort"+string+" is in NOT classpath!");
        return false;
    }
}

private void analyseImports() {
    Iterator it = katja.getKatjaImports();
    while (it.hasNext()) {
        Katja m = (Katja) it.next();
        Analyser analyser = new Analyser(m, this);
        analyser.analyse();
        num_errors += analyser.num_errors;
    }
}

private void fillDefinedHashes() {
    fillDefinedJavaImportHash(katja.getSpecification());
    fillDefinedSortIdHash(katja.getSpecification());
}

private void fillDefinedJavaImportHash(Specification spec) {
    ImportListIterator it = spec.importList().iterator();
    while (it.hasNext()) {
        Import imp = it.next();
        if (imp.is(JavaImport.sort)) {
            if (imp.is(JavaSimpleImport.sort)) {

```
String name = getNameOfImport(imp);
definedJavaImportHash.put(name, imp);
Katja.debug("Adding Java type "+name);
}
else
    error("Java imports on demand aren't supported yet!", imp.line());
}
}

/**
   * Returns the unqualified classname of the Import.<br>
   * E.g. for the import 'java.util.Iterator' it would return 'Iterator'.
   * @param imp the Import to return the classname of
   * @return the classname of the Import
   */
private String getNameOfImport(Import imp) {
    String name = imp.name().toString();
    int dot = name.lastIndexOf(".");
    name = name.substring(dot + 1, name.length());
    return name;
}

private void fillDefinedSortIdHash(Specification spec) {
    ProductionListIterator it = spec.prodList().iterator();
    while (it.hasNext()) {
        Production prod = it.next();
        if (definedSortIdHash.contains(prod.sortId().name())) {
            error("Sort "+prod.sortId().name()+" is already defined.", prod.sortId().line());
        } else {
            definedSortIdHash.add(prod.sortId().name());
            PackageDel pck = spec.pck();
            if (pck.is(Empty.sort)) {
                sortIdPackages.put(prod.sortId().name(), "");
                Katja.debug("Adding empty packagename to sort");
            } else {
                PackageName pckName = (PackageName) pck;
                sortIdPackages.put(prod.sortId().name().toString(), pckName.name().toString());
                Katja.debug("Adding packagename to sort: "+pckName.name());
            }
        }
        Katja.debug("Adding defined sort: "+prod.sortId().name());
    }
}

private void error(String string, KatjaInt line) {
    numErrors++;
    katja.addError(new KatjaError( 
        new KatjaString(string),
        new KatjaString(katja.getSpecFile()),
        line));
}

public int numberOfErrors() {
    return numErrors;
}
public HashMap getJavaImportHash() {
    return definedJavaImportHash;
}

KatjaElement.java

package katja.common;

public interface KatjaElement {
    public final static int sortInt = 1;
    public final static KatjaSort sort = new KatjaSort(KatjaElement.class, sortInt);
    public KatjaSort sort();
    public boolean is(KatjaSort s);
    public boolean eq(KatjaElement e);
    public boolean equals(Object o);
    public int hashCode();
    public String toString();
}

KatjaSort.java

package katja.common;

public final class KatjaSort extends KatjaElementImpl {
    public final static int sortInt = 5;
    public final static KatjaSort sort = new KatjaSort(KatjaSort.class, 5);
    private Class c1;
    private int id;
    protected Class getClassValue() {
        return c1;
    }
    public KatjaSort(Class c1, int id) {
        this.c1 = c1;
        this.id = id;
    }
    public KatjaSort sort() {
        return KatjaSort.sort();
    }
    public boolean eq(KatjaElement s) {
        if (! (s instanceof KatjaSort))
            return false;
        return this.id == ((KatjaSort)s).id;
    }
    public String toString() {
        return c1.getName();
    }
```java
public int toInt() {
    return id;
}

public int intValue() {
    return id;
}

public int hashCode() {
    return id;
}
```

### KatjaTerm.java

```java
package katja.common;

public interface KatjaTerm extends KatjaElement {
    public final static int sortInt = 2;
    public final static KatjaSort sort = new KatjaSort(KatjaTerm.class, sortInt);
    public int numSubterms();
    public int size();
}
```

### KatjaBuiltInType.java

```java
package katja.common;

public abstract class KatjaBuiltInType extends KatjaElementImpl implements KatjaTerm {
    public final static int sortInt = 6;
    public final static KatjaSort sort = new KatjaSort(KatjaBuiltInType.class, 6);
    public int numSubterms() {
        return 0;
    }
    public int size() {
        return 0;
    }
}
```

### KatjaTuple.java

```java
package katja.common;

public interface KatjaTuple extends KatjaTerm {
    public final static int sortInt = 3;
    public final static KatjaSort sort = new KatjaSort(KatjaList.class, 3);
}
```
public Object subterm(int ith);

KatjaList.java

package katja.common;

public interface KatjaList extends KatjaTerm{
    public final static int sortInt = 4;
    public final static KatjaSort sort = new KatjaSort(KatjaList.class,4);
}

KatjaElementImpl.java

package katja.common;

public abstract class KatjaElementImpl implements KatjaElement { 
    public boolean is(KatjaSort s) {
        return s.getClassValue().isInstance(this);
    }
    public boolean equals(Object o) {
        if (! (o instanceof KatjaElement))
            return false;
        return eq((KatjaElement) o);
    }
}

KatjaTermImpl.java

package katja.common;

public abstract class KatjaTermImpl extends KatjaElementImpl implements KatjaTerm {
    public int size() {
        return numSubterms();
    }
}

KatjaTupleImpl.java

package katja.common;

public abstract class KatjaTupleImpl extends KatjaTermImpl implements KatjaTuple {
}

KatjaListImpl.java

package katja.common;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public abstract class KatjaListImpl extends KatjaTermImpl implements KatjaTerm {
  protected List values = new ArrayList();

  protected abstract KatjaListImpl createInstance(List listValues);

  protected Object subtermInternal(int ith) {
    if (ith < 0)
      throw new KatjaIllegalArgumentException("Trying to access subterm 
        with index "+ ith+", but only values greater or equal to 0 are allowed.");
    if (ith >= values.size())
      return null;
    return values.get(ith);
  }

  public int numSubterms() {
    return values.size();
  }

  public int size() {
    return values.size();
  }

  protected boolean containsInternal(Object o) {
    return values.contains(o);
  }

  protected Object firstInternal() {
    if (values.size() == 0)
      return null;
    return values.get(0);
  }

  protected Object lastInternal() {
    if (values.size() == 0)
      return null;
    return values.get(values.size());
  }

  protected KatjaListImpl frontInternal() {
    if (values.size() == 0)
      return null;
    return createInstance(values.subList(0, values.size() - 1));
  }

  protected KatjaListImpl backInternal() {
    if (values.size() == 0)
      return null;
    return createInstance(values.subList(1, values.size()));
  }

  protected KatjaListImpl appFrontInternal(Object o) {

List list = new ArrayList(values.size()+1);
list.add(o);
list.addAll(values);

return createInstance(list);
}

protected KatjaListImpl removeInternal(Object o) {
    if (values.size() == 0)
        return null;

    List list = new ArrayList(values);
    list.remove(o);

    return createInstance(list);
}

protected KatjaListImpl removeAllInternal(KatjaListImpl l) {
    if (values.size() == 0)
        return null;

    List list = new ArrayList(values);
    list.removeAll(l.values);

    return createInstance(list);
}

protected KatjaListImpl appBackInternal(Object o) {
    List list = new ArrayList(values.size()+1);
    list.addAll(values);
    list.add(o);

    return createInstance(list);
}

protected KatjaListImpl conCInternal(KatjaListImpl l) {
    List list = new ArrayList(values.size()+l.values.size());
    list.addAll(values);
    list.addAll(l.values);

    return createInstance(list);
}

public boolean eq(KatjaElement e) {
    if (!(e instanceof KatjaListImpl))
        return false;

    KatjaListImpl el = (KatjaListImpl) e;
    return el.values.equals(values);
}

public int hashCode() {
    return values.hashCode()+this.getClass().hashCode()*31;
}

public String toString() {
    Iterator it = values.iterator();
    StringBuffer result = new StringBuffer();
    result.append("[");
    while (it.hasNext()) {
        result.append(it.next().toString());
    }
    result.append("]");
    return result.toString();
}
if (it.hasNext())
    result.append(".");
}
result.append("]");
return result.toString();

package katja.common;

public class KatjaString extends KatjaBuiltInType {
    public final static int sortInt = 9;
    public final static KatjaSort sort = new KatjaSort(KatjaString.class, sortInt);

    String value;

    public KatjaString(String value) {
        this.value = value;
    }

    public String toString() {
        return value;
    }

    public String stringValue() {
        return value;
    }

    public KatjaSort sort() {
        return KatjaString.sort;
    }

    public boolean eq(KatjaElement e) {
        if (! (e instanceof KatjaString))
            return false;
        return this.value.equals(((KatjaString)e).value);
    }

    public int hashCode() {
        return value.hashCode();
    }
}

package katja.common;

public class KatjaInt extends KatjaBuiltInType {
    public final static int sortInt = 8;
    public final static KatjaSort sort = new KatjaSort(KatjaInt.class, sortInt);
int value;

public KatjaInt ( int value ) {
    this.value = value;
}

public int toInt() {
    return value;
}

public int intValue() {
    return value;
}

public KatjaSort sort() {
    return KatjaInt.sort;
}

public boolean eq(KatjaElement e) {
    if (! (e instanceof KatjaInt))
        return false;
    return ((KatjaInt)e).toInt() == this.toInt();
}

public int hashCode() {
    return value;
}

public String toString() {
    return Integer.toString(value);
}

KatjaChar.java

package katja.common;

public class KatjaChar extends KatjaBuiltInType {
    public final static int sortInt = 7;
    public final static KatjaSort sort = new KatjaSort(KatjaChar.class, sortInt);

    char value;

    public KatjaChar(char value) {
        this.value = value;
    }

    public char toChar() {
        return value;
    }

    public char charValue() {
        return value;
    }

    public KatjaSort sort() {
        return KatjaChar.sort;
    }
}
```java
public boolean eq(KatjaElement e) {
    if (! (e instanceof KatjaChar))
        return false;
    return ((KatjaChar)e).value == this.value;
}

public int hashCode() {
    return value;
}

public String toString() {
    return Character.toString(value);
}
```

```java
package katja.common;

public class KatjaBool extends KatjaBuiltInType {
    public final static int sortInt = 10;
    public final static KatjaSort sort = new KatjaSort(KatjaBool.class, 10);

    public static final KatjaBool maxTrue = new KatjaBool(true);
    public static final KatjaBool maxFalse = new KatjaBool(false);

    boolean value;

    private KatjaBool(boolean value) {
        this.value = value;
    }

    public static KatjaBool fromBoolean(boolean b) {
        return b ? maxTrue : maxFalse;
    }

    public boolean toBoolean() {
        return value;
    }

    public boolean booleanValue() {
        return value;
    }

    public boolean eq(KatjaElement e) {
        return this == e;
    }

    public KatjaSort sort() {
        return KatjaBool.sort;
    }

    public int hashCode() {
        return value ? 5 : 31;
    }
}
```
public String toString() {
    return value ? "true" : "false";
}

KatjaUnknownTypeException.java

/ *
 * Copyright 2003
 * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
 * All rights reserved.
 */
package katja.common;

public class KatjaUnknownTypeException extends KatjaException {
    public KatjaUnknownTypeException(int typeConstant) {
        super("Unknown type: 1",typeConstant);
    }
}

KatjaIllegalMethodException.java

/ *
 * Copyright 2003, AG Software Technology, University of Kaiserslautern
 * Created on 08.05.2003
 */
package katja.common;

/** *
 * @author Jan Schaffer
 */
public class KatjaIllegalMethodException extends KatjaException {
    public KatjaIllegalMethodException(String msg) {
        super(msg);
    }
}

KatjaIllegalArgumentException.java

/ *
 * Copyright 2003
 * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
 * All rights reserved.
 */
package katja.common;

public class KatjaIllegalArgumentException extends KatjaException {
    public KatjaIllegalArgumentException(String msg) {
        super(msg);
    }
}
KatjaException.java

```java
package katja.common;

public class KatjaException extends RuntimeException {
    public KatjaException(String msg) {
        super(msg);
    }
}
```