Generating Order-Sorted Data Types in Java

Project Report

by

Jan Schäfer

February 5, 2004

AG Softwaretechnik
Fachbereich Informatik
Universität Kaiserslautern
Betreuer: Prof. Dr. Arnd Poetzsch-Heffter
Nicole Rauch

Contents

List of Figures **5**

List of Listings **5**

1 Introduction **6**

2 The Katja Specification Language **7**
2.1 General Structure 7
2.1.1 Package Name 7
2.1.2 Import Statements 7
2.2 Term Productions 8
2.2.1 Tuple Production 9
2.2.2 List Production 9
2.2.3 Variant Production 10
2.2.4 Nodes 10
2.3 Built-in Types 10
2.4 Using Java Classes 11
2.5 Comments 11
2.6 Example 11
2.7 Functions 11
2.7.1 General Functions 12
2.7.2 Term Functions 12
2.7.3 List Functions 12

3 The Java Interface **13**
3.1 Requirements 13
3.2 Type Hierarchy 13
3.2.1 Term Productions 14
3.2.2 Built-in Types 14
3.2.3 Sorts 15
3.2.4 Generated Classes 16
3.2.5 Variant Productions 17
3.3 Methods 18
3.3.1 Where Should the Methods Be Declared? 18
3.3.2 `KatjaElement` 19
3.3.3 `KatjaSort` 20
3.3.4 `KatjaTerm` 20
3.3.5 `KatjaTuple` 21
3.3.6 `KatjaList` 22
3.3.7 List Iterators 23

4 Implementation **24**
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

5 The Katja program **43**
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

6 Conclusions and Future Work **51**
6.1 Conclusions . . . 51
6.2 Future Work . . . 51

References **52**

A Migration from MAX to Katja **53**
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

A.4.4 Conversion of MAX Functions to Java 57
A.4.5 Conversion of MAX Java code to Katja Java code . 59
A.5 Conclusions 59

B The Source Code **60**

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.

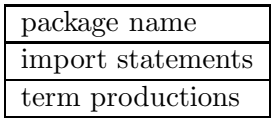

Figure 1: General structure of a KATJA specification file

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[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:

[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:

```
variantname = sort1 | sort2 | ... | 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 descibed 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.

```
/*
 * Katja specification file example
 */
package example
import java.utils.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.

Figure 2: One class for every term production

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).

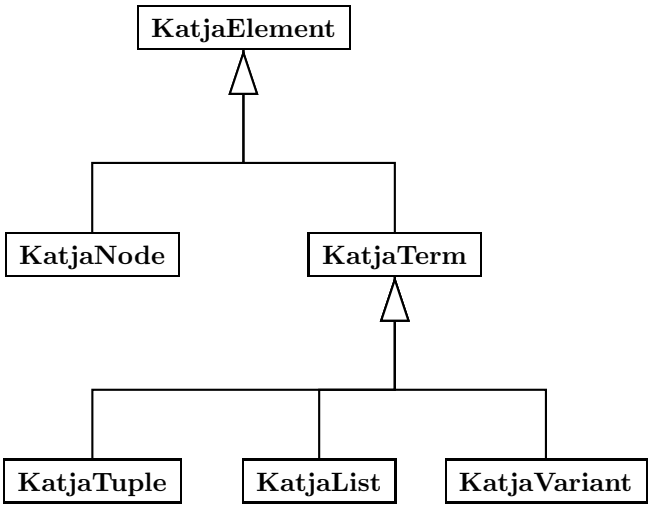

Figure 3: The classes `KatjaTerm`, `KatjaNode` and `KatjaElement`

Later there will be corresponding node classes for every term class.

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.

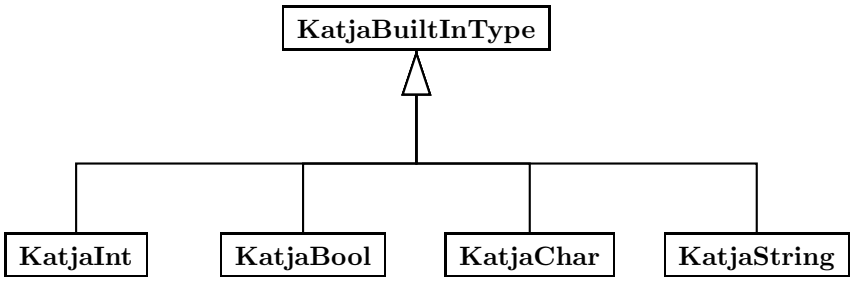

Figure 4: Built-in types

So the `KatjaBuiltInType` class has to inherit from `KatjaTerm` (see Figure 5).

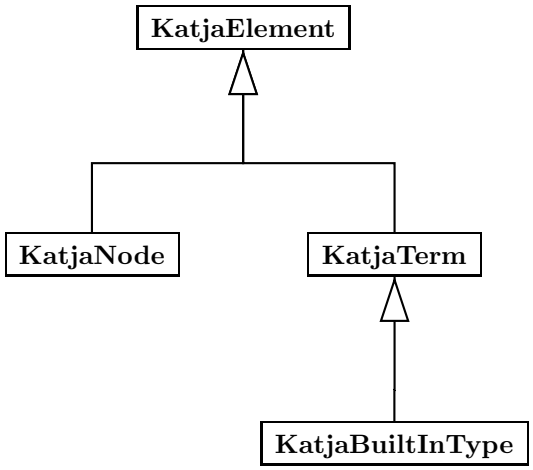

Figure 5: The `KatjaBuiltInType` class and the type hierarchy

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).

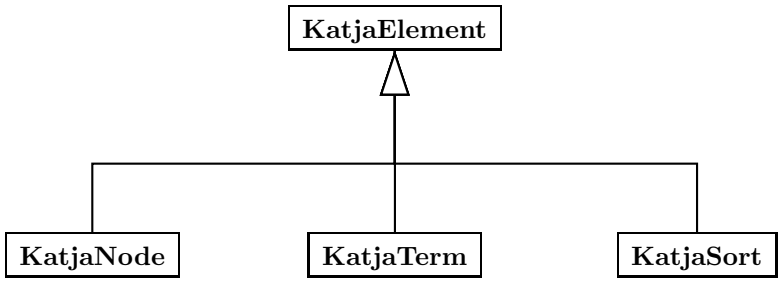

Figure 6: The `KatjaSort` class and the type hierarchy

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.

```
T ( L 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 `l`. `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).

Figure 7: All KATJA elements realized as single classes

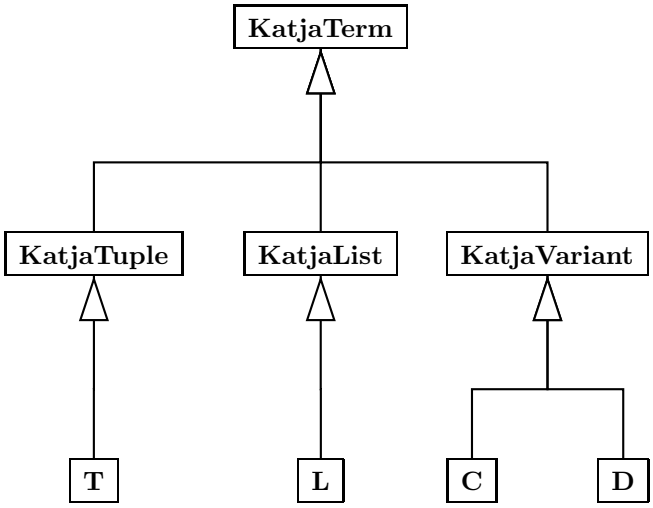

Figure 8: The generated classes inherit from their production classes

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 | 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)

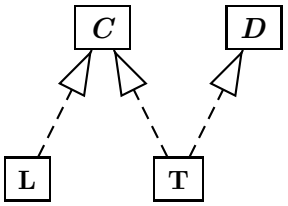

Figure 9: Variant productions implemented as interfaces

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

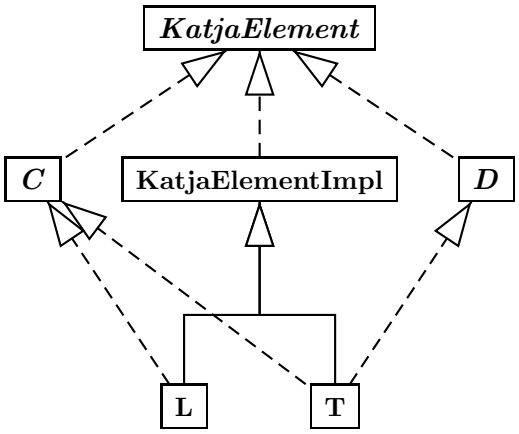

Figure 10: Variant productions and `KatjaElement`

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 wether 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).

```
// ... 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):

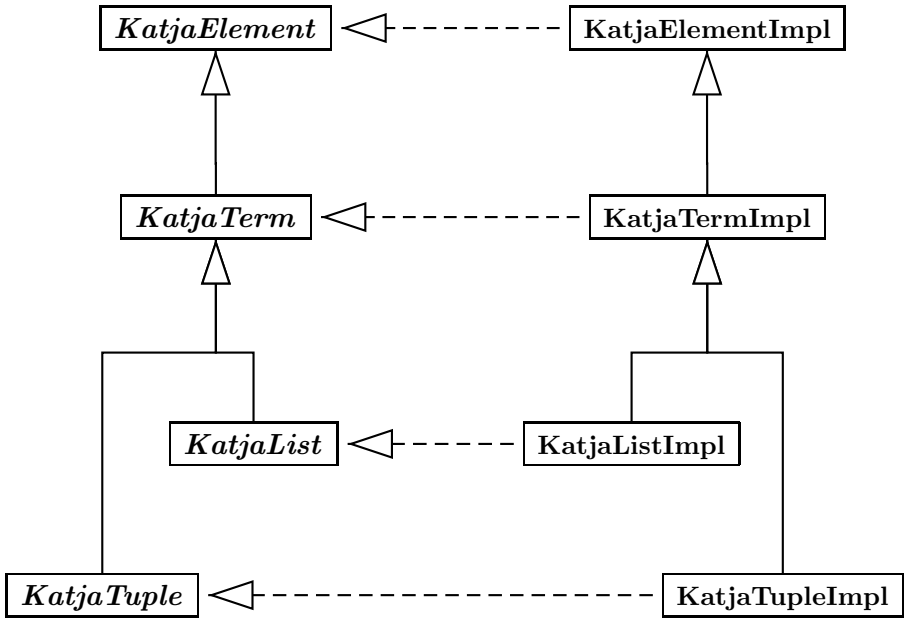

Figure 11: The two type hierarchies

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 $26^8 = 208,827,064,576 \approx 209$ Billion possibilities versus $2^{32} = 4,294,967,296 \approx$ 4,3 Billion different JAVA `int`s. 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 propability 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 `int`s 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 propability 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 propability 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 propability 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[3].

[3]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.

```
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.

```
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);
    }
    // ...
}
```

Listing 5: Implementation of the `is` function with the `isInstance` method

You may have noticed that it is now necessary that `T` implements the interfaces `C` and `D`, as they are variants of the type `T`. The variant implementation is discussed later in Section 4.4.3.

As this is now a general implementation of the `is` method that is independent of the class `T`, this function can be put into a super-class, namely `KatjaElementImpl`. That class also reimplements the `equals(Object o)` of `Object` by using the `eq` method. So it is possible to use KATJA classes within JAVA collection classes e.g. `java.util.Hashtable`. The `hashCode()` method, however, is implemented in the generated classes themselves. Listing 6 shows the `KatjaElementImpl` class.

```
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);
    }
}
```

Listing 6: The `KatjaElementImpl` class

4.2.2 The `KatjaSort` class

The `KatjaSort` class itself is rather simple. It contains the two mentioned attributes, a constructor to initialize them, and some getter methods. It also contains two `final static` attributes, which are contained in every KATJA class: `sortInt` which holds the sort constant and `sort` which holds the `KatjaSort` instance of the class.

You might wonder if it is necessary to have an extra `sortInt` attribute, as the value of the sort constant is also stored in the `KatjaSort` class. As I mentioned above, the JAVA `case` statement expects values that are known at compile time. So there has to be the extra `sortInt` attribute. It would, for example, not work to use `KatjaSort.sort.intValue()` in a `case` statement. This redundancy can be accepted because the values are generated and so

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:

```
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.

```
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

```
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)  {
```

```
        return this == e;
    }
    public KatjaSort sort() {
        return KatjaBool.sort;
    }
    public int hashCode() {
        return value ? 5 : 31;
    }
    public String toString() {
        return value ? "true" : "false";
    }
}
```

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:

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

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 $\mathcal{O}(1)$.

But there is another way to implement them. As the code is generated, and the number of subterms of a tuple is fix, 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.

```
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

```
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

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

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

Listing 12: The `KatjaTermImpl` class

```
public abstract class KatjaTupleImpl
       extends KatjaTermImpl
           implements KatjaTuple
{
}
```

Listing 13: The `KatjaTupleImpl` class

```
public class T
       extends KatjaTupleImpl
       implements C, D
{
    public static final int sortInt = 1950446000;
    public static 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;
```

```
            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, `ArrayList` is much faster than the `LinkedList`, because `LinkedList` has to create a new object for each of its entries.

The methods `front` and `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 `ArrayList` class of 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 `appBack` and `appFront` methods without copying the whole list in some cases. The `front` and `back` methods can be efficiently implemented by the `subList` method of the `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 JAVA `List` interface, which JAVA programmers are used to. These methods are: `get` instead of `subterm`, `size` instead of `numSubterms`, `add` instead of `appBack` and `addAll` instead of `conc`. Additionally I added some methods for convenience reasons: `remove(Object o)` that removes the first occurrence of the object o, `removeAll(Object o)` that removes all occurrences of the object o, and `contains(Object o)` which returns `true` if the object o is contained in the list and `false` otherwise.

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

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

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

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

Listing 15: The `KatjaList` interface

The current `KatjaListImpl` implementation is just a wrapper around the `ArrayList` class of JAVA. An important difference, however, is that instead of throwing an `IndexOutOfBoundsException`, `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);
```

```
    }
    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).

```
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:

```
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.

```
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 example[4] :

[4]This 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 `Expection` 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 by, however, situation 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).

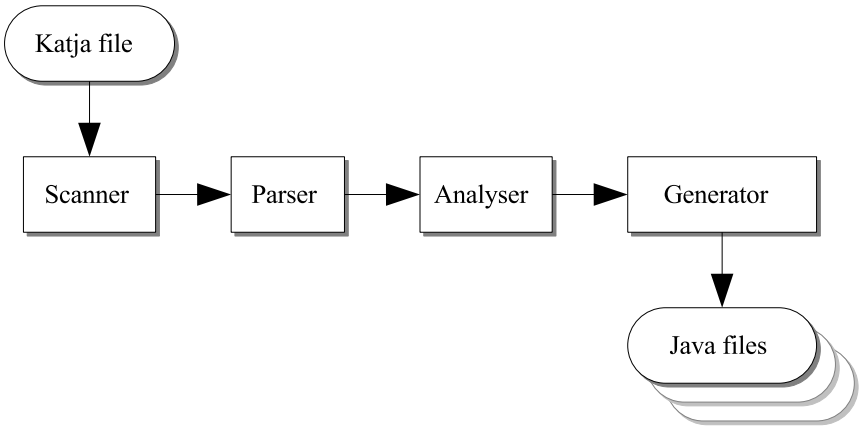

Figure 12: The four main parts of KATJA

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[5] 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).

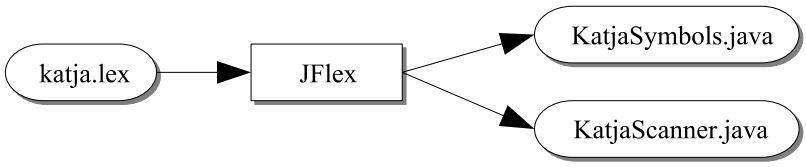

Figure 13: JFlex generates two files and has one as input.

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.

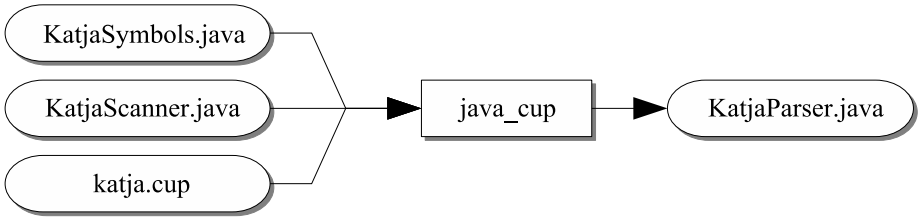

Figure 14: CUP has three files as input and one as output.

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.

```
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 )
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:

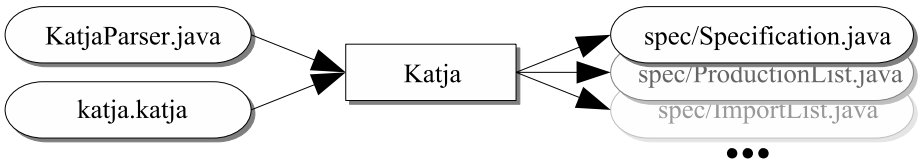

Figure 15: KATJA generating files from `katja.katja` using the parser.

- 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[6] file, which is called `katja.jar`.

To run KATJA the following programs have to be installed:

- At least JRE 1.3[7]
- JFlex [3]
- JAVA CUP [1]

[6] JAVA Archive
[7] The JRE can be downloaded from `http://java.sun.com`

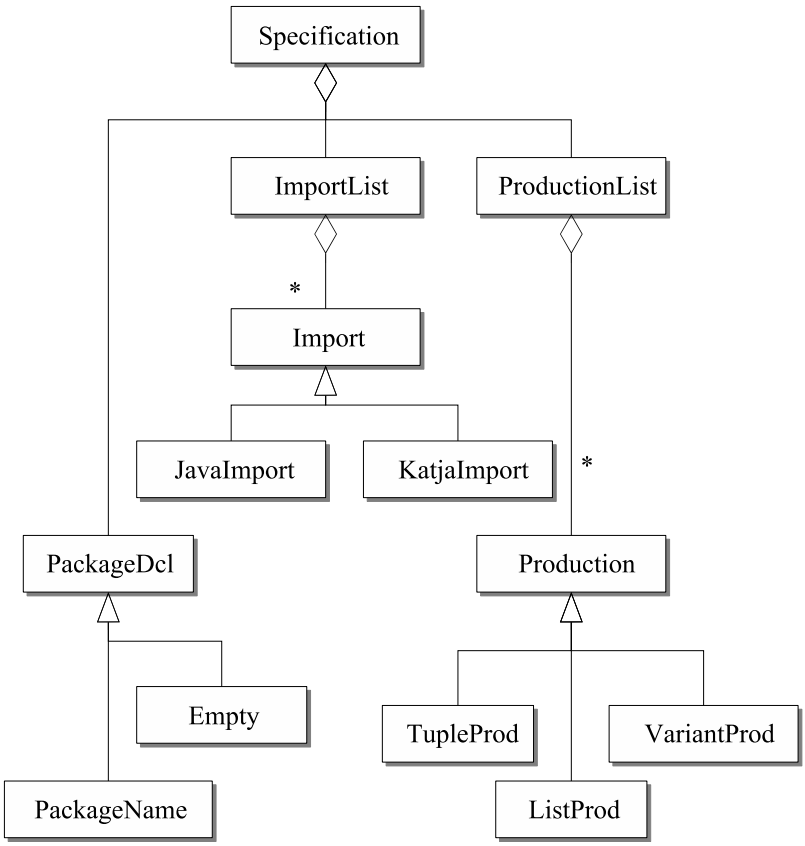

Figure 16: The `Specification` class and a selection of some other classes generated from the `katja.katja` file.

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>`.

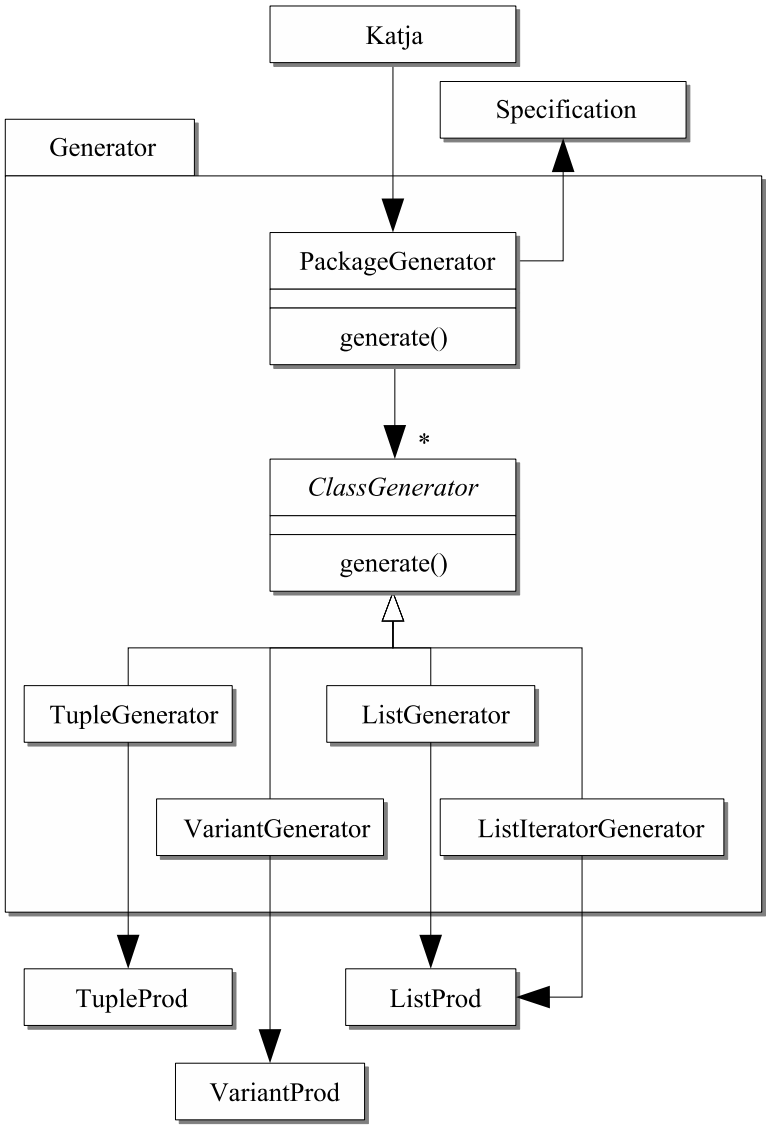

Figure 17: The generator classes

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

[1] CUP Parser Generator for Java, 1999. URL `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[3] Gerwin Klein. JFlex - The Fast Scanner Generator for Java , September 2003. URL `http://www.jflex.de/`.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[5] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX System. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, LNCS 714, pages 137–150. Springer-Verlag, 1993.

[6] A. Poetzsch-Heffter. Specification and prototyping of programming languages using the max system. In P. H. Schmidt, editor, *Logik in der Informatik: 3.Jahrestagung der GI-Fachgruppe 0.1.6, Bericht Nr. 23/95*, pages 85–94. Universität Karlsruhe, 1995. (Zusammenfassung einer Erweiterung von [7]).

[7] A. Poetzsch-Heffter and T. Eisenbarth. The MAX system: A tutorial introduction. Technischer Bericht TUM-I9307, Technische Universität München, April 1993.

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[10]. So the JAVA interface is just a bunch of `public static` methods defined in one class. The only type is a `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 `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 `Bool` or `Int` the JAVA types `boolean` and `int` are used. The KATJA `is` method for example returns a JAVA `boolean` instead of a MAX Bool. The call of a conversion function like `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 `is` method with anything other than a `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 `LD_LIBRARY_PATH` variable, and there is no need of loading this library.

The only thing you have to do is to add the `katja.jar` file and the KATJA-generated JAVA classes to your `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.

[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 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:

```
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:

```
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:

```
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:

```
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:

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

Now the KATJA file was finished and I let KATJA generate the JAVA classes:

```
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:

```
/* 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:

```
/* 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)
```

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:

```
/* 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:

```
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:

```
/* 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:

```
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)

B The Source Code

katja.lex

```
1  package katja;
2
3  import java_cup.runtime.Symbol;
4  import java.io.File;
5  %%
6  %class Scanner
7  %cupsym Symbols
8  %cup
9  %unicode
10 %line
11 %column
12 %{
13    StringBuffer string = new StringBuffer();
14    String filename;
15
16    public void setFilename(String filename) {
17            this.filename=filename;
18    }
19
20    public String getFile(String filename) {
21                  System.out.println("Include Datei: '"+filename+"'");
22                  if (filename.equals(""))
23                          return "";
24                  if (filename.length() < 10)
25                          return "";
26
27                  System.out.println(filename.length());
28
29          filename = filename.substring(8).trim();
30          System.out.println("Living1");
31                  if (filename.length() < 2)
32                          return "";
33          filename = filename.substring(1,filename.length()-1);
34                  System.out.println("Includete Datei: '"+filename+"'");
35                  System.out.println("Datei :"+this.filename);
36          File file = new File(this.filename);
37
38          String sep = System.getProperty("file.separator");
39          System.out.println("Living");
40          return file.getParent()+sep+filename;
41
42    }
43
44    private Symbol symbol(int type) {
45      return new Symbol(type, yyline, yycolumn);
46    }
47    private Symbol symbol(int type, Object value) {
48      return new Symbol(type, yyline, yycolumn, value);
49    }
50 %}
51 LineTerminator = \r|\n|\r\n
52 InputCharacter = [^\r\n]
53 WhiteSpace = {LineTerminator} | [ \t\f]
54 Comment = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}
55 TraditionalComment = "/*" [^*] ~"*/" {LineTerminator}*
56 EndOfLineComment = "//" {InputCharacter}* {LineTerminator}
57 DocumentationComment = "/**" ~"*/" {LineTerminator}*
```

```
58 Identifier = [:jletter:][:jletterdigit:]*
59
60 %%
61 "*"                                                { return symbol(Symbols.MULT); }
62 "("                                                { return symbol(Symbols.LBRACE)
       ; }
63 ")"                                                { return symbol(Symbols.RBRACE)
       ; }
64 "="                                                { return symbol(Symbols.EQUAL); }
65 "|"                                                { return symbol(Symbols.PIPE); }
66 ","                                                { return symbol(Symbols.COMMA); }
67 "."                                                { return symbol(Symbols.DOT); }
68 "import"                                    { return symbol(Symbols.IMPORT); }
69 "package"                              { return symbol(Symbols.PACKAGE); }
70 {Identifier}                           { return symbol(Symbols.IDENTIFIER, yytext()); }
71 {Comment}                              { /* ignore */ }
72 {WhiteSpace}                    { /* ignore */ }
73 . { System.err.println("Illegal character: "+yytext()+" in line "+(yyline+1)+", "+
       column "+(yycolumn+1)); }
```

katja.cup

```
1  package katja;
2
3  import katja.spec.*;
4  import katja.common.*;
5  import java_cup.runtime.Symbol;
6
7
8  action code {:
9          void error(String message, int line, int column) {
10                 System.out.println("Syntax Error in file '"+parser.filename+"', "+
       line "+(line+1)+", column "+(column+1)+":"+message);
11                 parser.number_of_errors++;
12          }
13 :}
14
15 parser code {:
16
17         String filename;
18         Specification spec;
19         Katja katja;
20
21         int number_of_errors = 0;
22
23         public int numberOfErrors() {
24                 return number_of_errors;
25         }
26
27
28         public Specification getSpecification() {
29                 return spec;
30         }
31
32         public void setKatja(Katja katja) {
33                 this.katja = katja;
34                 this.filename = katja.getSpecFile();
35         }
36
37         public void setFilename(String filename) {
```

```
38                  this.filename = filename;
39          }
40
41      public void report_error(String message, Object info) {
42
43          /* Create a StringBuffer called 'm' with the string 'Error' in it. */
44          StringBuffer m = new StringBuffer("\nError");
45          m.append(" in file '").append(filename).append("'");
46
47          /* Check if the information passed to the method is the same
48             type as the type java_cup.runtime.Symbol. */
49          if (info instanceof java_cup.runtime.Symbol) {
50              /* Declare a java_cup.runtime.Symbol object 's' with the
51                 information in the object info that is being typecasted
52                 as a java_cup.runtime.Symbol object. */
53              java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
54  //            KatjaLexer lexer = (KatjaLexer) this.getScanner();
55    //          m.append(" regarding text: '"+lexer.yytext()+"' ");
56
57              /* Check if the line number in the input is greater or
58                 equal to zero. */
59              if (s.left >= 0) {
60                  /* Add to the end of the StringBuffer error message
61                     the line number of the error in the input. */
62                  m.append(", line "+(s.left+1));
63                  /* Check if the column number in the input is greater
64                     or equal to zero. */
65                  if (s.right >= 0)
66                      /* Add to the end of the StringBuffer error message
67                         the column number of the error in the input. */
68                      m.append(", column "+(s.right+1));
69              }
70          }
71
72          /* Add to the end of the StringBuffer error message created in
73             this method the message that was passed into this method. */
74          m.append(" : "+message);
75
76          /* Print the contents of the StringBuffer 'm', which contains
77             an error message, out on a line. */
78          katja.error(m.toString());
79      }
80
81      /* Change the method report_fatal_error so when it reports a fatal
82          error it will display the line and column number of where the
83          fatal error occurred in the input as well as the reason for the
84          fatal error which is passed into the method in the object
85          'message' and then exit.*/
86      public void report_fatal_error(String message, Object info) {
87                  done_parsing();
88          report_error(message, info);
89      }
90
91          public void syntax_error(Symbol cur_token) {
92          number_of_errors++;
93                  report_error("Syntax error",cur_token);
94          }
95
96          public void  unrecovered_syntax_error(Symbol cur_token) {
97                  katja.error("Fatal Error : Couldn't repair and continue parse");
98                  done_parsing();
99          }
```

```
100
101 :}
102
103 terminal MULT, LBRACE, RBRACE, EQUAL, PIPE, COMMA, DOT;
104 terminal IMPORT, PACKAGE;
105 terminal java.lang.String IDENTIFIER;
106
107 non terminal Specification katja_spec;
108 non terminal ProductionList production_list;
109 non terminal Production production;
110 non terminal SortIdList pipe_list;
111 non terminal TupleParamList tuple_param_list;
112 non terminal TupleParam tuple_param;
113 non terminal PackageDcl package_declaration_opt;
114 non terminal PackageName package_declaration;
115 non terminal ImportList import_declarations_opt;
116 non terminal ImportList import_declaration_list;
117 non terminal Import import_declaration;
118 non terminal Import single_type_import_declaration;
119 non terminal Import type_import_on_demand_declaration;
120 non terminal KatjaString name;
121 non terminal KatjaString simple_name;
122 non terminal KatjaString qualified_name;
123 non terminal SortId sort_id;
124 non terminal TupleProd tuple_prod;
125 non terminal ListProd list_prod;
126 non terminal VariantProd variant_prod;
127 non terminal newlines;
128
129 katja_spec ::=
130                 package_declaration_opt:packagedcl
131                 import_declarations_opt:imports
132                 production_list:productions
133                 {:
134                         parser.spec = new Specification(packagedcl,imports,
            productions,new KatjaString(parser.filename));
135                 :}
136         ;
137
138 package_declaration_opt ::=
139                 package_declaration:name
140                 {:
141                         RESULT = name;
142                 :}
143         |
144                 {:
145                         RESULT = new Empty();
146                 :}
147         ;
148 import_declarations_opt ::=
149                 import_declaration_list:list
150                 {:
151                         RESULT = list;
152                 :}
153         |
154                 {:
155                         RESULT = new ImportList();
156                 :}
157         ;
158
159 production_list      ::=
160                 production:prod
```

```
161                 {:
162                         ProductionList list = new ProductionList();
163                         RESULT = list.appBack(prod);
164                 :}
165         |       production_list:list production:prod
166                 {:
167                         RESULT = list.appBack(prod);
168                 :}
169         ;
170
171 import_declaration_list ::=
172                 import_declaration:importdcl
173                 {:
174                         ImportList importList = new ImportList();
175                         RESULT = importList.appBack(importdcl);
176                 :}
177         |       import_declaration_list:list import_declaration:importdcl
178                 {:
179                         RESULT = list.appBack(importdcl);
180                 :}
181         ;
182
183 package_declaration ::=
184                 PACKAGE name:id
185                 {:
186                     Katja.debug("Package declaration "+id+" found !");
187                     RESULT = new PackageName(id);
188                 :}
189
190         ;
191
192 import_declaration ::=
193                 single_type_import_declaration:importdcl
194                 {:
195                         RESULT = importdcl;
196                 :}
197         |       type_import_on_demand_declaration:importdcl
198                 {:
199                         RESULT = importdcl;
200                 :}
201         ;
202
203 single_type_import_declaration ::=
204                 IMPORT name:id
205                 {:
206                         Katja.debug("Import declaration "+id+" found !");
207                         if (id.toString().endsWith(".katja"))
208                                 RESULT = new KatjaImport(id,new KatjaInt(idleft
            +1));
209                         else
210                                 RESULT = new JavaSimpleImport(id,new KatjaInt(
            idleft+1));
211                 :}
212
213         ;
214
215 type_import_on_demand_declaration ::=
216                 IMPORT name:id DOT MULT
217                 {:
218                         Katja.debug("Import declaration "+id+" found !");
219                         RESULT = new JavaOnDemandImport(new KatjaString(id+".*"),
            new KatjaInt(idleft+1));
```

```
220                     :}
221             |       IMPORT:im DOT MULT
222                     {:
223                             error("Missing import name after import keyword!",imleft
                    +1,imright+1);
224                             RESULT = new JavaOnDemandImport(new KatjaString(""),new
                    KatjaInt(imleft+1));
225                     :}
226             ;
227
228     sort_id ::=
229             name:id
230             {:
231                     if (Katja.isReservedWord(id.toString())) {
232                             error("The identifier '"+id+"' is a reserved word "+
233                                     "in Java and cannot be used as Katja identifier!"
                    ,idleft+1,idright+1);
234                     }
235
236                     if (Katja.isNativeType(id.toString())) {
237                             error("The identifier '"+id+"' is a Java native type "+
238                                     "and cannot be used as Katja identifier!",idleft
                    +1,idright+1);
239                     }
240
241                     String s = Katja.getBuiltInName(id.toString());
242                     RESULT = new SortId(new KatjaString(s), new KatjaInt(idleft+1));
243             :}
244             ;
245
246     name ::=
247                     simple_name:id
248                     {:
249                             RESULT = id;
250                     :}
251             |       qualified_name:id
252                     {:
253                             RESULT = id;
254                     :}
255             ;
256
257     simple_name ::=
258                     IDENTIFIER:id
259                     {:
260                             RESULT = new KatjaString(id);
261                     :}
262             ;
263
264     qualified_name ::=
265                      name:n DOT IDENTIFIER:id
266                     {:
267                             RESULT = new KatjaString(n+"."+id);
268                     :}
269             ;
270
271     production ::=
272                     tuple_prod:prod
273                     {:
274                             RESULT = prod;
275                     :}
276             |       list_prod:prod
277                     {:
```

```
278                             RESULT = prod;
279                     :}
280             |       variant_prod:prod
281                     {:
282                             RESULT = prod;
283                     :}
284             ;
285
286     tuple_prod ::=
287                     sort_id:id LBRACE tuple_param_list:list RBRACE
288                     {:
289                             Katja.debug("Tuple Production "+id.subterm(0)+" found !")
                    ;
290                             RESULT = new TupleProd(id,list,new KatjaInt(parser.katja.
                    getFreeSortInt()));
291                     :}
292             |       sort_id:id tuple_param_list:list RBRACE
293                     {:
294                             error("Missing left bracket before tuple production
                    parameters!",idleft+1,idright+1);
295                             RESULT = new TupleProd(id,list,new KatjaInt(parser.katja.
                    getFreeSortInt()));
296                     :}
297
298             ;
299
300     tuple_param_list                ::=
301                     tuple_param_list:list COMMA tuple_param:param
302                     {:
303                             RESULT = list.appBack(param);
304                     :}
305             |       tuple_param:param
306                     {:
307                             TupleParamList list = new TupleParamList();
308                             RESULT = list.appBack(param);
309                     :}
310             |
311                     {:
312                             RESULT = new TupleParamList();
313                     :}
314             ;
315
316     tuple_param ::=
317                     sort_id:typeId simple_name:sel
318                     {:
319                             RESULT = new Selector(typeId,sel);
320                     :}
321             |       sort_id:typeId
322                     {:
323                             RESULT = typeId;
324                     :}
325             |       sort_id:param name name
326                     {:
327                             error("Missing comma between tuple production parameters!
                    ",paramleft+1,paramright+1);
328                             RESULT = param;
329                     :}
330             |       sort_id:id name:param name name
331                     {:
332                             error("Missing comma between tuple production parameters!
                    ",paramleft+1,paramright+1);
333                             RESULT = id;
```

```
334                 :}
335
336         ;
337
338 list_prod ::=
339                 sort_id:id MULT sort_id:childType
340                 {:
341                         Katja.debug("List Production "+id.subterm(0)+" found !");
342                         RESULT = new ListProd(id,childType,new KatjaInt(parser.
            katja.getFreeSortInt()));
343                 :}
344                 ;
345 variant_prod ::=
346                 sort_id:id EQUAL pipe_list:list
347                 {:
348                         Katja.debug("Variant Production "+id.subterm(0)+" found !
            ");
349                         RESULT = new VariantProd(id,list,new KatjaInt(parser.
            katja.getFreeSortInt()));
350                 :}
351         ;
352
353 pipe_list ::=
354                 pipe_list:list PIPE sort_id:typeName
355                 {:
356                         RESULT = list.appBack(typeName);
357                 :}
358         |       sort_id:typeName
359                 {:
360                         SortIdList list = new SortIdList();
361                         RESULT = list.appBack(typeName);
362                 :}
363         ;
```

katja.katja

```
1  /*
2   * This is the Katja specification of the Katja language.
3   *
4   */
5  package katja.spec
6
7  Specification       ( PackageDcl pck, ImportList importList, ProductionList
               prodList, String filename)
8  PackageDcl          = PackageName | Empty
9  PackageName         ( String name)
10 ImportList          * Import
11 Import              = JavaImport | KatjaImport
12 JavaImport          = JavaSimpleImport | JavaOnDemandImport
13 JavaSimpleImport    ( String name, Int line )
14 JavaOnDemandImport  ( String name, Int line )
15 KatjaImport           ( String name, Int line )
16 ProductionList      * Production
17 Production          = TupleProd | VariantProd | ListProd
18 TupleProd           ( SortId sortId, TupleParamList params, Int sortInt )
19 TupleParamList      * TupleParam
20 TupleParam          = SortId | Selector
21 Selector            ( SortId sortId, String selector )
22 VariantProd         ( SortId sortId, SortIdList params, Int sortInt )
23 ListProd            ( SortId sortId, SortId param, Int sortInt )
```

```
24 SortIdList          * SortId
25 SortId              ( String name, Int line )
26 Empty               ( )
27
28 /*
29  * Helper classes
30  */
31 SelectorList        * Selector
32 KatjaError           ( String message, String file, Int line )
33 KatjaErrorList       * KatjaError
34 KatjaWarning         ( String message, String file, Int line )
35 KatjaWarningList     * KatjaWarning
```

Katja.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7
8  import java.io.File;
9  import java.io.FileNotFoundException;
10 import java.io.FileReader;
11 import java.io.IOException;
12 import java.util.HashMap;
13 import java.util.HashSet;
14 import java.util.Iterator;
15 import java.util.LinkedList;
16 import java.util.List;
17 import java.util.Random;
18
19 import katja.common.KatjaInt;
20 import katja.common.KatjaString;
21 import katja.spec.*;
22
23 /**
24  * The main class of the Katja system
25  * @author Jan Sch&auml;fer
26  */
27 public class Katja {
28
29     private static String VERSION = "0.1";
30     /**
31      * Wether debug messages should be shown
32      */
33     private static boolean DEBUG = false;
34
35     /**
36      * Wether any text should be shown on the
37      * standard output
38      */
39     private static boolean QUIET = false;
40
41     /**
42      * Wether code should be generated
43      */
44     private static boolean NOGEN = false;
45
```

```
46      /**
47       * Maps built-in names to there Java name.
48       * E.g. String will be KatjaString, Int will be KatjaInt and so on
49       */
50      private static HashMap builtInNames = new HashMap();
51
52      /**
53       * contains all Java reserved words.
54       */
55      private static HashSet reservedWords = new HashSet();
56
57      /**
58       * contains all Java native types
59       */
60      private static HashSet nativeTypes = new HashSet();
61
62      static {
63          builtInNames.put("String","KatjaString");
64          builtInNames.put("Int","KatjaInt");
65          builtInNames.put("Bool","KatjaBool");
66          builtInNames.put("Char","KatjaChar");
67          builtInNames.put("KatjaString","KatjaString");
68          builtInNames.put("KatjaInt","KatjaInt");
69          builtInNames.put("KatjaBool","KatjaBool");
70          builtInNames.put("KatjaChar","KatjaChar");
71
72          reservedWords.add("private");
73          reservedWords.add("abstract");
74          reservedWords.add("break");
75          reservedWords.add("case");
76          reservedWords.add("catch");
77          reservedWords.add("class");
78          reservedWords.add("const");
79          reservedWords.add("continue");
80          reservedWords.add("default");
81          reservedWords.add("do");
82          reservedWords.add("else");
83          reservedWords.add("extends");
84          reservedWords.add("final");
85          reservedWords.add("finally");
86          reservedWords.add("for");
87          reservedWords.add("future");
88          reservedWords.add("generic");
89          reservedWords.add("goto");
90          reservedWords.add("if");
91          reservedWords.add("implements");
92          reservedWords.add("import");
93          reservedWords.add("inner");
94          reservedWords.add("instanceof");
95          reservedWords.add("interface");
96          reservedWords.add("native");
97          reservedWords.add("new");
98          reservedWords.add("null");
99          reservedWords.add("operator");
100         reservedWords.add("outer");
101         reservedWords.add("package");
102         reservedWords.add("private");
103         reservedWords.add("protected");
104         reservedWords.add("public");
105         reservedWords.add("rest");
106         reservedWords.add("return");
107         reservedWords.add("static");
```

```
108             reservedWords.add("super");
109             reservedWords.add("switch");
110             reservedWords.add("synchronized");
111             reservedWords.add("this");
112             reservedWords.add("throw");
113             reservedWords.add("throws");
114             reservedWords.add("transient");
115             reservedWords.add("try");
116             reservedWords.add("var");
117             reservedWords.add("void");
118             reservedWords.add("volatile");
119             reservedWords.add("while");
120
121             nativeTypes.add("int");
122             nativeTypes.add("boolean");
123             nativeTypes.add("char");
124             nativeTypes.add("long");
125             nativeTypes.add("byte");
126             nativeTypes.add("double");
127             nativeTypes.add("float");
128             nativeTypes.add("short");
129
130         }
131
132         /**
133          * Contains all java imports.<br>
134          * The keys are String objects with the classnames and the
135          * values are String objects with the full qualified import.<br>
136          * E.g.: for 'java.util.Iterator' the key would be
137          * 'Iterator' and the value would be 'java.util.Iterator'<br>
138          * All imports of all imported and the current Katja file are
139          * stored.
140          */
141         HashMap javaImportHash;
142
143         /**
144          * Contains for every MAX SortId the package where
145          * it is declared.<br>
146          * The Keys and Values are String objects
147          */
148         HashMap sortIdPackageHash;
149
150         /**
151          *  Stores the sort integer for the next sort.
152          *  non-internal types start with a random number
153          *  that is at least 1000 big, so that it doesn't get into
154          *  the MAX built-in types sort integers.
155          *  <p>
156          *  The probability, that types of two different generated
157          *  MAX sorts have the same sortInt is then 1:2147482
158          */
159         private int nextsortint = ((new Random()).nextInt((Integer.MAX_VALUE-1000)
                    /1000))*1000;
160
161         /**
162          * Contains the Specification term created by the KatjaParser
163          */
164         private Specification spec;
165
166         /**
167          * Stores for every SortId that is contained in
168          * at least one Variant, a SortIdList which contains all
```

```
169         * Variants the SortId is contained in.
170         * Keys are the names of the SortIds
171         */
172        HashMap variantHash;
173
174        /**
175         * The directory where the classes should be generated
176         */
177        String destDir = null;
178
179        /**
180         * The name of the Katja specification file to parse
181         */
182        String specFile = null;
183
184        /**
185         * Contains the start time of Katja
186         */
187        private long startmillis;
188
189        /**
190         * Contains all error messages
191         */
192        private KatjaErrorList katjaErrorList;
193        private KatjaWarningList katjaWarningList;
194
195        /**
196         * A List of Katja instances of all
197         * imported Katja files
198         */
199        private List katjaImports;
200
201        /**
202         * If this file is imported from another specification
203         * this stores the Katja instance of that specification.
204         * Otherwise it is <code>null</code>
205         */
206        private Katja parentKatja;
207
208
209
210        /**
211         * Creates a new Katja instance with the given specification
212         * file and a Katja parent instance.<br>
213         * The desination directory will be taken from the
214         * parent Katja instance.<br>
215         * The nextsortint will be setted to the nextsortint of
216         * the parent instance.<br>
217         * A parent is a Katja instance which specification
218         * file imports the specifcation of this instance.
219         * @param specFile the specification file to parse
220         * @param parentKatja the parent Katja instance which specification
221         * file imports specFile
222         */
223        public Katja(String specFile, Katja parentKatja) {
224            this.specFile = specFile;
225            this.destDir = parentKatja.destDir;
226            this.nextsortint = parentKatja.nextsortint;
227            this.javaImportHash = parentKatja.javaImportHash;
228            this.sortIdPackageHash = parentKatja.sortIdPackageHash;
229            this.variantHash = parentKatja.variantHash;
230            this.parentKatja = parentKatja;
```

```
231     }
232
233     /**
234      * Creates a new Katja instance with the given specification
235      * file. The classes are generated to the
236      * current working directory
237      * @param specFile the path of the specifcation file
238      * @throws Exception if an error occurres
239      */
240     public Katja(String specFile) throws Exception {
241         this(specFile,"");
242     }
243
244     /**
245      * Creates a new Katja instance with the given specification
246      * file. The classes are generated to the given
247      * destination directory. If destDir is empty or <code>null</code>
248      * the current working directory is taken
249      * @param _specFile the path of the specification file
250      * @param _destDir the directory where the classes should
251      * be generated to. If it is empty or <code>null</code>
252      * the current working directory is taken
253      * @throws Exception if an error occures
254      */
255     public Katja(String _specFile, String _destDir) throws Exception {
256         destDir = _destDir;
257         specFile = _specFile;
258
259         if (destDir == null || destDir == "") {
260             destDir = System.getProperty("user.dir");
261         }
262
263         javaImportHash = new HashMap();
264         sortIdPackageHash = new HashMap();
265         variantHash = new HashMap();
266         katjaErrorList = new KatjaErrorList();
267         katjaWarningList = new KatjaWarningList();
268     }
269
270     /**
271      * Starts parses the specification file, analyses it and generates
272      * the classes
273      * @throws FileNotFoundException if the specification file
274      * or the destination directory doesn't exist
275      * @throws Exception if an error occures
276      */
277     public void parse()
278         throws FileNotFoundException, Exception
279     {
280         startmillis = System.currentTimeMillis();
281         showCopyright();
282         checkFileExistence();
283
284         outputln("Specification file: "+this.specFile);
285         outputln("Destination directory: "+this.destDir);
286
287         output("Parsing ... ");
288
289         Parser parser = parseFile();
290
291         if (parser.numberOfErrors() == 0) {
292             outputln("done");
```

```
293             output("Analysing ... ");
294             fillVariantHashtable();
295             Analyser analyser = analyse();
296             if (analyser.numberOfErrors() == 0) {
297                 outputln("done");
298                 output("Generating code ... ");
299                 if (NOGEN) {
300                     outputln("skipped");
301                 } else {
302                     generateCode();
303                     outputln("done");
304                 }
305 
306 
307                 if (katjaWarningList.size()>0) {
308                     showWarnings();
309                     outputln("Warnings: "+katjaWarningList.size());
310                 }
311                 outputln("\nBUILD SUCCESSFUL");
312                 showTotalTime(startmillis);
313             }
314             else {
315                 outputln("failed");
316                 showBuildFailedAndExit();
317             }
318         } else {
319             outputln("failed");
320             showBuildFailedAndExit();
321         }
322 
323     }
324 
325     private Analyser analyse() {
326         Analyser analyser = new Analyser(this);
327         analyser.analyse();
328         return analyser;
329     }
330 
331     private void generateCode() throws IOException {
332         generateImportsCode();
333         PackageGenerator pkgGen = new PackageGenerator(this,destDir);
334         pkgGen.generate();
335     }
336 
337     /**
338      * Generates the code of all imported
339      * Katja specifications
340      * @throws IOException if a problem during the writing of the files occurs
341      */
342     private void generateImportsCode() throws IOException {
343         Iterator it = katjaImports.iterator();
344         while(it.hasNext()) {
345             ((Katja)it.next()).generateCode();
346         }
347     }
348 
349     /**
350      * Only parses the specification file, but doesn't
351      * analyse or generate something
352      * @return the KatjaParser instance of the parser
353      * @throws FileNotFoundException if the
354      * @throws Exception
```

```
355         */
356        private Parser parseFile() throws FileNotFoundException, Exception {
357            Scanner lexer = new Scanner(new FileReader(specFile));
358            lexer.setFilename(specFile);
359
360            Parser parser = new Parser(lexer);
361            parser.setKatja(this);
362            parser.parse();
363
364            if (parser.numberOfErrors() == 0) {
365                spec = parser.getSpecification();
366                parseImports();
367            }
368
369            return parser;
370        }
371
372
373        private void checkFileExistence() {
374            if (! (new File(destDir)).exists()) {
375                error("The destination directory '"+destDir+"' doesn't exist !\n");
376                showBuildFailedAndExit();
377            }
378
379            if (! (new File(destDir)).canWrite()) {
380                error("You have no write permissions for the destination directory '"
                    +destDir+"' !\n");
381                showBuildFailedAndExit();
382            }
383
384            if (!(new File(specFile)).exists()) {
385                error("The specified file '"+specFile+"' doesn't exist!\n");
386                showBuildFailedAndExit();
387            }
388        }
389
390        /**
391         * Creates a new KatjaError and adds it to katjaErrorList
392         *
393         * @param errormsg the message of the error
394         * @param file the file where the error have been occurred
395         * @param line the line of the error
396         */
397        public void error(String errormsg, String file, KatjaInt line) {
398            addError(new KatjaError( new KatjaString(errormsg),
399                new KatjaString(file), line));
400        }
401
402        /**
403         * Creates a new KatjaError with the given message
404         * and adds it to katjaErrorList
405         * @param errormsg the error message
406         */
407        public void error(String errormsg) {
408            addError(new KatjaError( new KatjaString(errormsg),
409                new KatjaString(""), new KatjaInt(-1)));
410        }
411
412        /**
413         * Prints all error messages stored
414         * in katjaErrorList
415         */
```

```
416     private void showErrors() {
417         KatjaErrorListIterator it = katjaErrorList.iterator();
418         while (it.hasNext()) {
419             KatjaError e = it.next();
420             String msg = "Error ";
421             if (! e.file().toString().equals(""))
422                 msg += "in file "+e.file();
423 
424             if (! e.line().eq(new KatjaInt(-1))) {
425                 msg += ":"+e.line();
426             }
427 
428             msg += ":"+e.message();
429             outputln(msg);
430         }
431     }
432 
433     /**
434      * Prints all warn messages stored
435      * in katjaWarningList
436      */
437     private void showWarnings() {
438         KatjaWarningListIterator it = katjaWarningList.iterator();
439         while (it.hasNext()) {
440             KatjaWarning e = it.next();
441             String msg = "Warning ";
442             if (! e.file().toString().equals(""))
443                 msg += "in file "+e.file();
444 
445             if (! e.line().eq(new KatjaInt(-1))) {
446                 msg += ":"+e.line();
447             }
448 
449             msg += ":"+e.message();
450             outputln(msg);
451         }
452     }
453 
454     /**
455      * Shows all warnings and errors and exits with
456      * result 1
457      */
458     private void showBuildFailedAndExit() {
459         showWarnings();
460         showErrors();
461         outputln("\nWarnings: "+katjaWarningList.size());
462         outputln("Errors: "+katjaErrorList.size()+"\n");
463         outputln("BUILD FAILED");
464         showTotalTime(startmillis);
465         System.exit(1);
466     }
467 
468     /**
469      * Parses all imported Katja files
470      */
471     private void parseImports() throws FileNotFoundException, Exception {
472         ImportList impList = spec.importList();
473         ImportListIterator it = impList.iterator();
474         katjaImports = new LinkedList();
475         while (it.hasNext()) {
476             Import imp = it.next();
477             if (imp.is(KatjaImport.sort)) {
```

```
478                 String importFile = ((KatjaImport)imp).name().toString();
479                 debug("Importing file "+importFile);
480                 File f = new File(specFile);
481                 File impFile = new File(f.getParent(),importFile);
482                 if ( fileAlreadyParsed(impFile.getPath())) {
483                     error("Cyclic import of file "+importFile+"!",specFile,imp.
            line());
484                 } else {
485                     Katja m = new Katja(impFile.getPath(),this);
486                     m.parseFile();
487                     katjaImports.add(m);
488                     this.nextsortint = m.nextsortint;
489                 }
490             }
491         }
492     }
493
494     /**
495      * Returns wether the given Katja file gets
496      * parsed in a parent Katja file already
497      * @param filePath the file to check
498      * @return wether the given Katja file gets
499      * parsed in a parent Katja file already
500      */
501     private boolean fileAlreadyParsed(String filePath) {
502         if (specFile.equals(filePath))
503             return true;
504
505         if (parentKatja != null) {
506             return parentKatja.fileAlreadyParsed(filePath);
507         }
508
509         return false;
510     }
511
512     /**
513      * Does a <code>System.out.println</code> if
514      * not in quiet mode
515      * @param string the string to print out
516      */
517     public static void outputln(String string) {
518         if (QUIET)
519             return;
520
521         System.out.println(string);
522
523     }
524
525     /**
526      * Does a <code>System.out.print</code> if
527      * not in quiet mode
528      * @param string the string to print out
529      */
530     public static void output(String string) {
531         if (QUIET)
532             return;
533
534         System.out.print(string);
535
536     }
537
538     public String getSpecFile() {
```

```
539         return specFile;
540     }
541 
542     /**
543      * Returns a <code>SortIdList</code> containing all
544      * <code>SortId</code>'s of variants where sortId
545      * is contained in.
546      * @param sortId the <code>SortId</code> to get the list of.
547      * @return a <code>SortIdList</code> containing all
548      * <code>SortId</code>'s of variants where sortId
549      * is contained in
550      */
551     public SortIdList getVariantsOfSortId(SortId sortId) {
552         return (SortIdList) variantHash.get(sortId.name());
553     }
554 
555     /**
556      * Fills the variantHash field
557      */
558     private void fillVariantHashtable() {
559         ProductionListIterator it = spec.prodList().iterator();
560         while (it.hasNext()) {
561             Production prod = it.next();
562             if (prod.is(VariantProd.sort)) {
563                 addVariantToHashtable((VariantProd) prod);
564             }
565         }
566 
567         fillVariantHashtableOfKatjaImports();
568     }
569 
570     private void fillVariantHashtableOfKatjaImports() {
571         Iterator katjaIt = getKatjaImports();
572         while (katjaIt.hasNext()) {
573             Katja m = (Katja) katjaIt.next();
574             m.fillVariantHashtable();
575         }
576     }
577 
578     /**
579      * Adds a VariantPod to the variantHash Hashtable.<br>
580      * This means for every SortId contained in prod it adds
581      * prod to the SortIdList of the Hashtable with the key
582      * of the SortId
583      * @param prod the VariantProd to add to the variantHash Hashtable
584      */
585     private void addVariantToHashtable(VariantProd prod) {
586         SortIdListIterator it = prod.params().iterator();
587         while (it.hasNext()) {
588             SortId sortId = it.next();
589             SortIdList variantList = (SortIdList) variantHash.get(sortId.name());
590             int hashCode = sortId.hashCode();
591 
592             if (variantList == null) {
593                 variantList = new SortIdList();
594             }
595             variantHash.put(sortId.name(), variantList.appBack(prod.sortId()));
596             debug("Adding "+prod.sortId().name()+" to "+sortId.name()+" hash: "+
                hashCode);
597         }
598     }
599 
```

```
600     /**
601      * Prints the copyright note to the standard output
602      * if the QUIET flag is not set
603      */
604     private static void showCopyright() {
605         if (QUIET)
606             return;
607 //        System.out.println();
608         System.out.println("------------------------------------------------------------");
609         System.out.println("              Katja v"+VERSION);
610         System.out.println(" (c) copyright 2003 AG Softwaretechnology");
611         System.out.println(" University of Kaiserslautern , Germany");
612         System.out.println("------------------------------------------------------------");
613         System.out.println();
614     }
615
616     /**
617      * Prints the total amount of time needed
618      * to the standard output if the QUIET flag
619      * is not set.
620      * @param startMillis the milliseconds of the start time
621      */
622     private void showTotalTime(long startMillis) {
623         if (QUIET)
624             return;
625         long endMillis = System.currentTimeMillis();
626         long diffMillis = endMillis-startMillis;
627         long hours = (diffMillis/3600000);
628         long minutes = (diffMillis/60000) % 60;
629         long seconds = (diffMillis/1000) % 60;
630         long millis = diffMillis % 1000;
631
632         System.out.print("Total time: ");
633         if (hours > 0)
634             System.out.print(hours+" hours ");
635
636         if (minutes > 0)
637             System.out.print(minutes+" minutes ");
638
639         if (seconds > 0)
640             System.out.print(seconds+" seconds ");
641
642         if (millis > 0)
643             System.out.print(millis+" ms");
644
645         System.out.println();
646         System.out.println();
647     }
648
649     /**
650      * Prints the usage to the standard output and exit(1) after that
651      * if the QUIET flag is not set
652      */
653     private static void showUsageAndExit() {
654         if (! QUIET) {
655             showCopyright();
656             System.out.println("Usage: java katja.Katja [options] <filename>");
657             System.out.println();
658             System.out.println("Options:");
659             System.out.println(" -h -help ........show this help");
660             System.out.println(" -d=<dir>");
```

```
661             System.out.println("  -dest=<dir>    generate the Java classes to <dir
                >.");
662             System.out.println("                 If this is not set, the classes
                are generated to ");
663             System.out.println("                 the current working directory.");
664             System.out.println("  -q -quiet      be quiet, suppress any output");
665             System.out.println("  -debug         show debug messages");
666             System.out.println("  -nogen         only analyse, but don't generate
                anything");
667             System.out.println("  -sortint=<n>   start sort integer constants at <
                n>");
668             System.out.println();
669         }
670         System.exit(1);
671     }
672
673     /**
674      * The main function
675      * @param args the arguments
676      */
677     public static void main(String[] args) throws Exception {
678         String destDir = null;
679         String specFile = null;
680
681         if (args.length < 1) {
682             System.out.println("\nNo filename specified!\n");
683             showUsageAndExit();
684         }
685
686         int startSortInt = 0;
687
688         for (int i=0; i<args.length; i++) {
689             if (args[i].equals("-q") ||
690                 args[i].equals("-quiet"))
691                 Katja.QUIET=true;
692             else
693             if (args[i].equals("-debug"))
694                 Katja.DEBUG=true;
695             else
696             if (args[i].equals("-nogen"))
697                 Katja.NOGEN=true;
698             else
699             if (args[i].equals("-help") ||
700                 args[i].equals("-h"))
701                 showUsageAndExit();
702             else
703             if (args[i].startsWith("-d=") ||
704                 args[i].startsWith("-dest="))
705                 destDir = getParamValue(args[i]);
706             else
707             if (args[i].startsWith("-sortint="))
708                 startSortInt = Integer.parseInt(getParamValue(args[i]));
709             else
710             if (args[i].startsWith("-")) {
711                 outputln("Unkown option: "+args[i]+"\n");
712                 showUsageAndExit();
713             }
714         }
715
716         specFile = args[args.length-1];
717         if (specFile.startsWith("-")) {
718             outputln("\nNo filename specified!\n");
```

```
719             showUsageAndExit();
720         }
721
722         Katja katja = new Katja(specFile, destDir);
723
724         if (startSortInt > 0) {
725             if (startSortInt < 100) {
726                 outputln("\nThe starting sort integer has to be greater or equal
                    to 100");
727                 showUsageAndExit();
728             } else
729                 katja.nextsortint = startSortInt;
730         }
731
732         katja.parse();
733     }
734
735
736     /**
737      * Returns the value of an option.<br>
738      * E.g. for the option '-dest=/tmp' it would return '/tmp'
739      * @param string an option
740      * @return value of an option
741      */
742     private static String getParamValue(String string) {
743         return string.substring(string.indexOf("=")+1,string.length());
744     }
745
746     public static void debug(String s) {
747         if (DEBUG)
748             System.out.println(s);
749     }
750
751     public Specification getSpecification() {
752         return spec;
753     }
754
755     /**
756      * Checks wether the given String s is a built-in name
757      * such as String or Int and returns the Java name for
758      * it like KatjaString or KatjaInt.
759      * <p>
760      * If the given String s is not a built-in name it is returned
761      * without changing anything.
762      * @param s the String to get the Java name for
763      * @return a String that can be used
764      */
765     public static String getBuiltInName(String s) {
766         String s2 = (String) builtInNames.get(s);
767         if (s2 == null)
768             return s;
769         return s2;
770     }
771
772     /**
773      * Returns wether the given String s is a reserved word in Java
774      * @param s the String to check
775      * @return wether the given String s is a reserved word in Java
776      */
777     public static boolean isReservedWord(String s) {
778         return reservedWords.contains(s);
779     }
```

```
780
781     /**
782      * Returns wether the given String s is a Java native type
783      * @param s the String to check
784      * @return wether the given String s is a Java native type
785      */
786     public static boolean isNativeType(String s) {
787         return nativeTypes.contains(s);
788     }
789
790     /**
791      * Returns wether the given String s is a Katja built-in type
792      * like Int, Bool etc.
793      * @param s the String to check
794      * @return wether the given String s is a Katja built-in type
795      */
796     public static boolean isBuiltInType(String s) {
797         return builtInNames.containsValue(s);
798     }
799
800     /**
801      * Returns a free sort integer
802      * @return a free sort integer
803      */
804     public int getFreeSortInt() {
805         return nextsortint++;
806     }
807
808     /**
809      * Returns an Iterator of Katja objects of all imported Katja files
810      * @return an Iterator of Katja objects of all imported Katja files
811      */
812     public Iterator getKatjaImports() {
813         return katjaImports.iterator();
814     }
815
816     /**
817      * Returns the full qualified name of the given name
818      * @param name the Java type of which the full qualified name should be
             returned
819      * @return the full qualified name of the given name
820      */
821     public JavaImport getJavaImportForName(String name) {
822         return (JavaImport) javaImportHash.get(name);
823     }
824
825     /**
826      * Returns the package name of the given sort name
827      * @param sortName the sort name to get the package of
828      * @return the package name of the given sort name
829      */
830     public String getPackageOfSortId(String sortName) {
831         return (String) sortIdPackageHash.get(sortName);
832     }
833
834     /**
835      * Adds the given error to the internal error list
836      * @param error the error to add to the inernal error list
837      */
838     public void addError(KatjaError error) {
839         if (parentKatja == null)
840             katjaErrorList = katjaErrorList.add(error);
```

```
841             else
842                 parentKatja.addError(error);
843         }
844 
845         /**
846          * Adds the given warning to the internal warning list
847          * @param warning the warning to add to the internal warning list
848          */
849         public void addWarning(KatjaWarning warning) {
850             if (parentKatja == null)
851                 katjaWarningList = katjaWarningList.add(warning);
852             else
853                 parentKatja.addWarning(warning);
854         }
855 
856 }
```

PackageGenerator.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   *
6   * Created on 05.07.2003
7   */
8  package katja;
9 
10 import java.io.File;
11 import java.io.IOException;
12 
13 import katja.spec.ListProd;
14 import katja.spec.PackageDcl;
15 import katja.spec.PackageName;
16 import katja.spec.Production;
17 import katja.spec.ProductionList;
18 import katja.spec.TupleProd;
19 import katja.spec.VariantProd;
20 
21 /**
22  * The MaxPackageGenerator is responsible for creating
23  * a package containing generated classes.
24  * You initialize it with the package name and the
25  * destination directory where to generate the classes.
26  * After that you call the different methods to create the
27  * desired classes.
28  * @author Jan Sch&auml;fer
29  */
30 public class PackageGenerator {
31 
32     Katja katja;
33 
34     /**
35      * This variable holds the name of the
36      * package for all created classes
37      */
38     private String packageName = "";
39 
40     /**
41      * This variable holds the directory where
```

```
42          * the generated classes should be put in.
43          */
44         private File destDir;
45
46         /**
47          * This constructor creates a new <code>MaxClassGenerator</code>
48          * object an sets the package name and the destination directory
49          * where the generated classes should be put in.
50          * <p>
51          * Notice that the generatoed classes will be put under
52          * the destination directory plus the package directory.
53          * <p>
54          * E.g. if destDir is <i>/tmp/test</i> and the package name
55          * is <i>my.katja.test</i> the classes would be put into the
56          * directory <i>/tmp/test/my/katja/test</i>
57          * @param katja the Katja instance
58          * @param destDir the destination directory of the generated classes
59          */
60         public PackageGenerator(Katja katja, String destDir) throws IOException {
61             this.katja = katja;
62             PackageDcl packageDcl = katja.getSpecification().pck();
63             if (packageDcl.is(PackageName.sort))
64                 this.packageName = ((PackageName) packageDcl).name().stringValue();
65
66             this.destDir = new File(destDir,getPackageDirectory());
67             if ( ! this.destDir.exists()) {
68                  this.destDir.mkdirs();
69             }
70         }
71
72         /**
73          * Returns the directory of the package name.
74          * <p>
75          * E.g. if the package name is <i>my.katja.test</i> the
76          * result would be <i>my/katja/test</i> on a UNIX system
77          * resp. <i>my\katja\test</i> on a Windows(R) system
78          * @return the directory for the package name
79          */
80         protected String getPackageDirectory() {
81             return packageName.replace('.', File.separatorChar);
82         }
83
84         /**
85          * Return the directory where classes will be generated to,
86          * including the package directory
87          * @return the directory where classes will be generated to.
88          */
89         public File getDestinationDir() {
90             return destDir;
91         }
92
93         /**
94          * Returns the package name of the generated classes
95          * @return the package name of the generated classes
96          */
97         public String getPackageName() {
98             return packageName;
99         }
100
101        public void generate() throws IOException {
102            ProductionList prodList = katja.getSpecification().prodList();
103            for (int i=0; i<prodList.numSubterms(); i++) {
```

```
104             Production prod = prodList.subterm(i);
105             generate(prod);
106         }
107     }
108
109     private void generate(Production prod) throws IOException {
110         switch (prod.sort().toInt()) {
111             case TupleProd.sortInt : generate((TupleProd) prod); break;
112             case ListProd.sortInt : generate((ListProd) prod); break;
113             case VariantProd.sortInt : generate((VariantProd) prod); break;
114         }
115     }
116
117     private void generate(TupleProd prod) throws IOException {
118         (new TupleGenerator(this,prod)).generate();
119     }
120
121     private void generate(ListProd prod) throws IOException {
122         (new ListGenerator(this,prod)).generate();
123     }
124
125     private void generate(VariantProd prod) throws IOException {
126         (new VariantGenerator(this,prod)).generate();
127     }
128
129 }
```

ClassGenerator.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7
8  import java.io.BufferedWriter;
9  import java.io.File;
10 import java.io.FileWriter;
11 import java.io.IOException;
12 import java.io.PrintWriter;
13 import java.util.Iterator;
14 import java.util.LinkedHashSet;
15 import java.util.LinkedList;
16 import java.util.List;
17 import java.util.ListIterator;
18 import java.util.Set;
19 import java.util.Collection;
20
21 import katja.spec.JavaImport;
22 import katja.spec.Production;
23 import katja.spec.SortIdList;
24 import katja.spec.SortIdListIterator;
25
26 /**
27  * This class is the super class for the specialized
28  * generators. It implements common methods
29  * to generate Java classes.
30  * @author Jan Sch&auml;fer
31  */
```

```
32 public abstract class ClassGenerator {
33     /**
34      * Holds destination directory and package name
35      */
36     protected PackageGenerator pkgGen;
37 
38     /**
39      * Holds the name of the class to be generated
40      */
41     protected String className;
42 
43     /**
44      * The Production to create the class from
45      */
46    protected Production prod;
47 
48     /**
49      * The Writer to write the generated class
50      */
51     protected PrintWriter writer;
52 
53     /**
54      * Creates a new ClassGenerator.
55      * @param packageGenerator the PackageGenerator
56      * @param prod the Production to create the class from
57      * @author Jan Sch&auml;fer
58      */
59     public ClassGenerator(
60                 PackageGenerator packageGenerator,
61                 Production prod)
62     {
63         this.pkgGen = packageGenerator;
64         this.className = prod.sortId().name().stringValue();
65         this.prod = prod;
66     }
67 
68     /**
69      * Creates a new ClassGenerator.
70      * @param packageGenerator
71      * @param className the name of the class to be generated
72      */
73     public ClassGenerator(
74                 PackageGenerator packageGenerator,
75                 String className)
76     {
77         this.pkgGen = packageGenerator;
78         this.className = className;
79     }
80 
81     /**
82      * Generates the Java file
83      * @throws IOException if a problem during the writing of the file occurs
84      */
85     public void generate() throws IOException {
86         this.writer = new PrintWriter(
87                     new BufferedWriter(
88                         new FileWriter(getFilePath())));
89         generatePackageName();
90         generateImportList();
91         generateClassDeclaration();
92         generateClassBody();
93         this.writer.close();
```

```
94      }
95
96
97      /**
98       * Generates the package declaration of this class
99       * or none if it has no package name.
100      */
101     private void generatePackageName() {
102         if (this.pkgGen.getPackageName().equals(""))
103             return;
104
105         writer.println("package "+this.pkgGen.getPackageName()+";");
106         writer.println(); // Empty line after the package declaration
107     }
108
109     /**
110      * Generates the list of import declarations.
111      * It uses for this the List that is returned by the getImportList
112      * method
113      */
114     private void generateImportList() {
115         Iterator it = getImportList().iterator();
116
117         while (it.hasNext()) {
118             writer.println("import "+it.next()+";");
119         }
120
121         writer.println(); // Empty line after the import list
122     }
123
124     private void generateClassDeclaration() {
125         writer.print("public ");
126
127         if (isInterface())
128             writer.print("interface ");
129         else
130             writer.print("class ");
131
132         writer.print(className);
133         if (getExtendedClass() != null)
134             writer.print(" extends "+getExtendedClass()+" ");
135         generateImplementedInterfaces();
136         writer.println();
137     }
138
139     /**
140      * Generates the list of implemented interfaces.
141      * It uses for this the List returned by the getImplementedInterfaces
142      * method.
143      */
144     private void generateImplementedInterfaces() {
145         ListIterator it = getImplementedInterfaces().listIterator();
146         if (it.hasNext()) {
147             // if its an interface we have to extend the other interfaces
148             // instead of implementing them
149             if (isInterface())
150                 writer.print(",");
151             else
152                 writer.print("implements ");
153         }
154
155         while(it.hasNext()) {
```

```
156             writer.print(it.next());
157             if (it.hasNext())
158                 writer.print(", ");
159         }
160     }
161
162     /**
163      * Generates the body of the class.
164      */
165     private void generateClassBody() {
166         writer.println("{");
167         generateSortDefinition();
168         generateMemberVariables();
169         generateMethods();
170         generateSortMethod();
171         writer.println("}");
172     }
173
174     /**
175      * Generates the KatjaSort related static members:
176      * <pre>
177      * public static int sortInt = ...
178      * public static KatjaSort sort = ...
179      * </pre>
180      */
181     protected void generateSortDefinition() {
182         // For interfaces there is no static sortInt defined, because it
183         // is never needed.
184         if (!isInterface())
185             writer.println("    public static final int sortInt = "+prod.sortInt
                ()+";");
186         writer.println("    public static final KatjaSort sort = new KatjaSort("+
                className+".class,"+prod.sortInt()+");");
187         writer.println();
188
189     }
190
191     /**
192      * Generates all member variables.<br>
193      * Has to be overriden by specialized class
194      * generators.
195      */
196     protected abstract void generateMemberVariables();
197
198     /**
199      * Generates all methods.<br>
200      * Has to be overriden by specialized class
201      * generators.
202      */
203     protected abstract void generateMethods();
204
205     /**
206      * Generates the sort() method
207      */
208     protected void generateSortMethod() {
209         writer.print("    public KatjaSort sort()");
210         if (isInterface())
211             writer.println(";");
212         else {
213             writer.println(" {");
214             writer.println("        return "+className+".sort;");
215             writer.println("    }");
```

```
216             }
217         }
218 
219 
220         /**
221          * Returns the filename of this class
222          * @return the filename of this class
223          */
224         public String getFileName() {
225             return className+".java";
226         }
227 
228         /**
229          * Returns the path of the file of this class
230          * @return the path of the file of this class
231          */
232         public File getFilePath() {
233             return new File(this.pkgGen.getPackageDirectory(),getFileName());
234         }
235 
236         /**
237          * Returns the class name that should be extended by this
238          * class or <code>null</code> if the class doesn't extend
239          * any class.
240          * This method must be overriden by specialized
241          * generator classes.
242          * @return the class that should be extended by this class.
243          */
244         protected abstract String getExtendedClass();
245 
246         /**
247          * Returns a list of String objects, that contains the
248          * interface names of interfaces which should be implemented
249          * by this class. This method must be overriden by specialized
250          * generator classes. <br>
251          * When overide this method don't forget to call the super method
252          * to not lost the implemented interfaces of the super class
253          * @return
254          */
255         protected List getImplementedInterfaces() {
256             List interfaces = new LinkedList();
257 
258             if (prod != null) {
259                 SortIdList variantList = pkgGen.katja.getVariantsOfSortId(prod.sortId
                    ());
260                 if (variantList == null)
261                     return interfaces;
262 
263                 SortIdListIterator it = variantList.iterator();
264 
265                 while (it.hasNext()) {
266                     interfaces.add(it.next().name().toString());
267                 }
268             }
269 
270             return interfaces;
271         }
272 
273         /**
274          * Returns a list of String objects, that contains the
275          * full qualified imported classes. This method should
276          * be overidden in specialized generator classes.
```

```
277         * <p>
278         * Notice to call the super() method to not lose the
279         * imported classes of this class. Which is in fact <code>katja.common.*</
                code>
280         *
281         * @return a list of String objects, that contains the
282         * full qualified imported classes
283         */
284        protected Collection getImportList() {
285              Set result = new LinkedHashSet();
286              result.addAll(getImportListOfSortIdNames());
287              result.addAll(getImportListOfImplementedInterfaces());
288              return result;
289        }
290
291        private List getImportListOfImplementedInterfaces() {
292            Iterator it = getImplementedInterfaces().iterator();
293            LinkedList result = new LinkedList();
294            while (it.hasNext()) {
295                String name = (String) it.next();
296                if (name != null) {
297                    String qualifiedName = getFullQualifiedName(name);
298                    if (qualifiedName != null)
299                        result.add(qualifiedName);
300                }
301            }
302
303            return result;
304        }
305
306        /**
307         * Returns a List of String objects of full qualified
308         * classnames which are generated of the List returned by
309         * the getSortIdName method
310         * @return a List of String objects of full qualified
311         * classnames which are generated of the List returned by
312         * the getSortIdName method
313         */
314        private List getImportListOfSortIdNames() {
315            List result = new LinkedList();
316            Iterator it = getSortIdNames().iterator();
317
318            while (it.hasNext()) {
319                String sortIdName = (String) it.next();
320                String fullQualifiedName = getFullQualifiedName(sortIdName);
321                if (fullQualifiedName != null &&
322                    ! result.contains(fullQualifiedName))
323                    result.add(fullQualifiedName);
324            }
325
326            return result;
327        }
328
329        /**
330         * Returns the full qualified name of a classname<br>
331         * There are two possibilities to find out the full qualified
332         * name. Either the name is an import Java type, then the Java
333         * import statement is used, or the type is a Katja type then the
334         * package name plus the type is returned.<br>
335         * If nothing works and the full qualified name would be the
336         * same as the given parameter, <code>null</code> is returned
337         *
```

```
338         * E.g. for 'List' it would return 'java.util.List' iff
339         * there is an 'import java.util.List' statement in the Katja
340         * file.<br>
341         * for 'Exp' it would return 'mypackage.Exp' if there is an
342         * 'package mypackage' statement in the Katja file where Exp is
343         * defined.
344         *
345         * @param sortIdName the name to find out the full qualified name
346         * @return The full qualified name of the given name, or <code>null</code>
347         * if the returen name would be the same as the parameter
348         */
349        private String getFullQualifiedName(String sortIdName) {
350            // Test if it is a Java type
351            JavaImport imp = pkgGen.katja.getJavaImportForName(sortIdName);
352            if (imp != null) {
353                return imp.name().toString();
354            } else { // If no Java type look for the packagename
355                String pck = pkgGen.katja.getPackageOfSortId(sortIdName);
356                if (pck != null)
357                    Katja.debug(sortIdName+" sort has package: "+pck);
358                else
359                    Katja.debug(sortIdName+" sort has no package");
360
361                // If a package name is declared return the package name
362                // plus the sort name, otherwise return null
363                if (pck != null && !pck.equals("") && !pck.equals(pkgGen.
                   getPackageName())) {
364                    return pck+"."+sortIdName;
365                } else
366                    return null;
367            }
368        }
369
370        /**
371         * Returns a List of String objects of the
372         * names of sorts that are used in the subclass.<br>
373         * This List is then used to create the needed imports.
374         * @return a List of String objects of the
375         * names of sorts that are used in the subclass.
376         */
377        protected abstract List getSortIdNames();
378
379        /**
380         * Returns wether an interface rather a class should be generated<br>
381         * The default implementation returns false.<br>
382         * Specialized generators can override this to generate an interface
383         * rather a class
384         * @return wether an interface rather a class should be generated
385         */
386        protected boolean isInterface() {
387            return false;
388        }
389    }
```

TupleGenerator.java

```
1   /*
2    * Copyright 2003
3    * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4    * All rights reserved.
```

```
5   */
6  package katja;
7
8  import java.util.Collection;
9  import java.util.LinkedList;
10 import java.util.List;
11
12 import katja.common.KatjaString;
13 import katja.spec.Selector;
14 import katja.spec.SortId;
15 import katja.spec.TupleParam;
16 import katja.spec.TupleParamListIterator;
17 import katja.spec.TupleProd;
18
19 /**
20  * This class generates a class from a Katja tuple.
21  * @author Jan Sch&auml;fer
22  */
23 public class TupleGenerator extends ClassGenerator {
24
25     private TupleProd tupleProd;
26
27     /**
28      * Creates a new KatjaTupleGenerator
29      *
30      * @param packageGenerator the PackageGenerator
31      * @param prod the TupleProd to generate the class from
32      */
33     public TupleGenerator(
34                     PackageGenerator packageGenerator,
35                     TupleProd prod)
36     {
37         super(packageGenerator, prod);
38         this.tupleProd = prod;
39     }
40
41     /**
42      * Generates all member variables.
43      * Which are one variable for every child
44      *
45      * @see katja.ClassGenerator#generateMemberVariables()
46      */
47     protected void generateMemberVariables() {
48         if (tupleProd.params().numSubterms()==0)
49             return;
50
51         for (int i=0; i<tupleProd.params().numSubterms(); i++) {
52             TupleParam param = tupleProd.params().subterm(i);
53             SortId sortId = getSortIdFromTupleParam(param);
54
55             writer.println("    private "+sortId.subterm(0)+" child"+i+";");
56         }
57         writer.println();
58     }
59
60     private SortId getSortIdFromTupleParam(TupleParam param) {
61         SortId sortId;
62         if (param.is(Selector.sort))
63             sortId = ((Selector)param).sortId();
64         else
65             sortId = (SortId) param;
66         return sortId;
```

```
67      }
68
69      protected void generateMethods() {
70          generateConstructor();
71          generateNumSubterms();
72          generateGet();
73          generateSubterm();
74          generateEq();
75          generateSelectorFunctions();
76          generateHashcode();
77          generateToString();
78      }
79
80      private void generateConstructor() {
81          // We do not need an empty constructor
82          if (tupleProd.params().numSubterms() == 0)
83              return;
84          writer.print("    public "+className+"(");
85          generateConstructorArguments();
86          writer.println(") {");
87          generateConstructorBody();
88          writer.println("    }");
89          writer.println();
90
91      }
92
93      private void generateConstructorArguments() {
94          for (int i=0; i<tupleProd.params().numSubterms(); i++) {
95              TupleParam param = tupleProd.params().subterm(i);
96              SortId sortId = getSortIdFromTupleParam(param);
97
98              writer.print(sortId.name()+" arg"+i);
99              if (i<tupleProd.params().numSubterms()-1)
100                 writer.print(", ");
101         }
102     }
103
104     private void generateConstructorBody() {
105         for (int i=0; i<tupleProd.params().numSubterms(); i++) {
106             writer.println("        this.child"+i+" = arg"+i+";");
107         }
108     }
109
110     private void generateNumSubterms() {
111         writer.println("    public int numSubterms() {");
112         writer.println("        return "+tupleProd.params().numSubterms()+";");
113         writer.println("    }");
114         writer.println();
115     }
116
117     private void generateGet() {
118         writer.println("    public Object get(int ith) {");
119         writer.println("        return subterm(ith);");
120         writer.println("    }");
121         writer.println();
122     }
123
124     /**
125      * Generates the subterm method.<br>
126      * If the class has no subterms at all, this
127      * method isn't generated.
128      */
```

```
129     private void generateSubterm() {
130         // Don't generate the subterm method if the
131         // class has no subterms
132         //if (prod.params().numSubterms()==0)
133         //     return;
134         writer.println("    public Object subterm(int ith) {");
135 
136         writer.println("        switch (ith) {");
137 
138         for (int i=0; i<tupleProd.params().numSubterms(); i++) {
139             writer.println("            case "+i+": return _child"+i+";");
140         }
141 
142         writer.println("            default:");
143         writer.println("                if (ith<0) {");
144         writer.println("                    throw new
                KatjaIllegalArgumentException(");
145         writer.println("                        \"Calling the subterm method
                with negative parameter \"+ith+\" !\");");
146         writer.println("                } else {");
147         writer.println("                    throw new
                KatjaIllegalArgumentException(");
148         writer.println("                        \"Trying to access subterm
                \"+");
149         writer.println("                        ith+\", but sort '\"+sort().
                toString()+\"' has only "+
150                                                     +tupleProd.params().
                numSubterms()+" subterms!\");");
151         writer.println("                }");
152         writer.println("        }");
153         writer.println("    }");
154         writer.println();
155     }
156 
157     private void generateEq() {
158         writer.println("    public boolean eq(KatjaElement e) {");
159         writer.println("        if (!e.is("+className+".sort))");
160         writer.println("            return false;");
161         writer.println();
162 
163         if (tupleProd.params().numSubterms()>0) {
164             writer.println("        "+className+" t = ("+className+") e;");
165             writer.println();
166             for (int i=0; i<tupleProd.params().numSubterms(); i++) {
167                 writer.println("        if (!(t._child"+i+".equals(_child"+i+"))
                )");
168                 writer.println("            return false;");
169             }
170             writer.println();
171         }
172 
173         writer.println("        return true;");
174         writer.println("    }");
175         writer.println();
176     }
177 
178     private void generateSelectorFunctions() {
179         for (int i=0; i<tupleProd.params().numSubterms(); i++) {
180             TupleParam param = tupleProd.params().subterm(i);
181             if (param.is(Selector.sort)) {
182                 generateSelectorFunction((Selector)param,i);
183             }
```

```
184             }
185         }
186 
187         private void generateSelectorFunction(Selector selector,int ithArg) {
188             SortId sortId = selector.sortId();
189             KatjaString selName = selector.selector();
190             writer.print("    public ");
191             writer.print(sortId.name()+" ");
192             writer.println(selName+"() {");
193             writer.println("        return  child"+ithArg+";");
194             writer.println("    }");
195             writer.println();
196         }
197 
198         /**
199          * Generates the hashCode function
200          */
201         private void generateHashcode() {
202             writer.println("    public int hashCode() {");
203             writer.print("        return "+tupleProd.sortInt());
204             for (int i=0; i<tupleProd.params().numSubterms(); i++) {
205                 writer.println("+");
206                 writer.print("            child"+i+".hashCode()*31");
207             }
208 
209             writer.println(";");
210 
211             writer.println("    }");
212             writer.println();
213         }
214 
215         /**
216          * Generates the toString function
217          */
218         private void generateToString() {
219             writer.println("    public String toString() {");
220             writer.println("        StringBuffer result = new StringBuffer();");
221             writer.println("        result.append(\"( \");");
222             for (int i=0; i<tupleProd.params().numSubterms(); i++) {
223                 writer.println("        result.append( child"+i+".toString());");
224                 if (i+1 < tupleProd.params().numSubterms() )
225                     writer.println("        result.append(\", \");");
226             }
227 
228             writer.println("        result.append(\" )\");");
229             writer.println("        return result.toString();");
230             writer.println("    }");
231             writer.println();
232         }
233 
234 
235         /**
236          * Returns 'KatjaElementImpl'
237          * @see katja.ClassGenerator#getExtendedClass()
238          */
239         protected String getExtendedClass() {
240             return "KatjaTupleImpl";
241         }
242 
243         protected List getSortIdNames() {
244             List list = new LinkedList();
245 
```

```
246         TupleParamListIterator it = tupleProd.params().iterator();
247         while (it.hasNext()) {
248             TupleParam param = it.next();
249             SortId sortId;
250             if (param.is(SortId.sort))
251                 sortId = (SortId) param;
252             else
253                 sortId = ((Selector) param).sortId();
254 
255             list.add(sortId.name().toString());
256 
257         }
258 
259         return list;
260     }
261 
262     protected Collection getImportList() {
263         Collection col = super.getImportList();
264         col.add(new String("katja.common.*"));
265         return col;
266     }
267 }
```

VariantGenerator.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7 
8  import java.util.Collection;
9  import java.util.LinkedList;
10 import java.util.List;
11 
12 import katja.common.KatjaInt;
13 import katja.common.KatjaString;
14 import katja.spec.KatjaWarning;
15 import katja.spec.Production;
16 import katja.spec.ProductionList;
17 import katja.spec.ProductionListIterator;
18 import katja.spec.Selector;
19 import katja.spec.SelectorList;
20 import katja.spec.SelectorListIterator;
21 import katja.spec.SortId;
22 import katja.spec.SortIdList;
23 import katja.spec.SortIdListIterator;
24 import katja.spec.TupleParam;
25 import katja.spec.TupleParamListIterator;
26 import katja.spec.TupleProd;
27 import katja.spec.VariantProd;
28 
29 /**
30  * Create a class from a VariantProd
31  * @author Jan Sch&auml;fer
32  */
33 public class VariantGenerator extends ClassGenerator {
34 
35     private SortIdList children;
```

```
36      private VariantProd variantProd;
37      private SelectorList allSelectors = null; // Lazy initialisation
38
39      /**
40       * Creates a new KatjaVairiantGenerator
41       * @param packageGenerator the PackageGenerator
42       * @param prod the VariantProd to create the class from
43       */
44      public VariantGenerator(
45          PackageGenerator packageGenerator,
46          VariantProd prod)
47      {
48          super(packageGenerator, prod);
49          this.children =  prod.params();
50          this.variantProd = prod;
51      }
52
53      /**
54       * Does nothing, as interfaces have no member variables
55       * @see katja.ClassGenerator#generateMemberVariables()
56       */
57      protected void generateMemberVariables() {
58      }
59
60      /**
61       * Generates all SelectorFunctions that are the same in all subtypes
62       * @see katja.ClassGenerator#generateMethods()
63       */
64      protected void generateMethods() {
65          generateSelectorFunctions();
66      }
67
68      private void generateSelectorFunctions() {
69          SelectorListIterator it = getAllSelectors().iterator();
70
71          while(it.hasNext()) {
72              generateSelectorFunction(it.next());
73          }
74      }
75
76      private void generateSelectorFunction(Selector selector) {
77          SortId sortId = selector.sortId();
78          KatjaString selName = selector.selector();
79          writer.print("    public ");
80          writer.print(sortId.name()+" ");
81          writer.println(selName+"();");
82          writer.println();
83      }
84
85      private SelectorList getAllSelectors() {
86          if (allSelectors == null)
87              allSelectors = getAllSelectors(variantProd);
88
89          return allSelectors;
90      }
91
92
93      private SelectorList getAllSelectors(VariantProd variant) {
94          ProductionListIterator it = getAllProductions(variant).iterator();
95          SelectorList list = new SelectorList();
96
97          // First add all Selectors of the first TupleProduction
```

```
98          if (it.hasNext()) {
99              Production p = it.next();
100             if (p.is(TupleProd.sort)) {
101                 list = getAllSelectors((TupleProd) p);
102             } else if (p.is(VariantProd.sort)) {
103                 list = getAllSelectors((VariantProd) p);
104             } else {
105                 // If there is a ListProduction there are no Selectors
106                 return new SelectorList();
107             }
108         }
109
110         // Then merge the others
111         while (it.hasNext()) {
112             Production p = it.next();
113             if (p.is(TupleProd.sort)) {
114                 list = mergeSelectorLists(list,getAllSelectors((TupleProd) p));
115             } else if (p.is(VariantProd.sort)) {
116                 list = mergeSelectorLists(list,getAllSelectors((VariantProd) p));
117             } else {
118                 // If there is a ListProduction there are no Selectors
119                 return new SelectorList();
120             }
121         }
122         return list;
123     }
124
125     private ProductionList getAllProductions(VariantProd variant) {
126         ProductionListIterator it = pkgGen.katja.getSpecification().prodList().
                iterator();
127         ProductionList list = new ProductionList();
128         while (it.hasNext()) {
129             Production p = it.next();
130             if (variantContainsSortId(variant,p.sortId())) {
131                 list = list.add(p);
132             }
133         }
134         return list;
135     }
136
137     private boolean variantContainsSortId(VariantProd variant, SortId id) {
138         SortIdListIterator it = variant.params().iterator();
139         while(it.hasNext()) {
140             SortId sortId = it.next();
141             if (sortId.name().eq(id.name()))
142                 return true;
143         }
144         return false;
145     }
146
147     private SelectorList getAllSelectors(TupleProd p) {
148         TupleParamListIterator it = p.params().iterator();
149         SelectorList result = new SelectorList();
150         while(it.hasNext()) {
151             TupleParam param = it.next();
152             if (param.is(Selector.sort)) {
153                 result = result.add((Selector) param);
154             }
155         }
156         return result;
157     }
158
```

```
159
160     private SelectorList mergeSelectorLists(SelectorList a, SelectorList b) {
161         SelectorListIterator it = a.iterator();
162         SelectorList result = new SelectorList();
163         while (it.hasNext()) {
164             Selector sel = it.next();
165             if (selectorListContains(b,sel))
166                 result = result.add(sel);
167             else {
168                 Selector sel2 = getSelectorWithName(b, sel.selector());
169                 if (sel2 != null) {
170                     pkgGen.katja.addWarning(new KatjaWarning(
171                         new KatjaString("Conflict between types of selectors!\n"+
172                             "Can't add selector '"+sel.selector()+"' to variant '
        "+className+"' !"),
173                         new KatjaString(""), new KatjaInt(-1)));
174                 }
175
176                 continue;
177             }
178         }
179         return result;
180     }
181
182     private boolean selectorListContains(SelectorList b, Selector sel) {
183         SelectorListIterator it = b.iterator();
184         while (it.hasNext()) {
185             Selector sel2 = it.next();
186             if (sel2.selector().eq(sel.selector())) {
187                 if (sel2.sortId().name().eq(sel.sortId().name())) {
188                     return true;
189                 }
190             }
191         }
192         return false;
193     }
194
195     private Selector getSelectorWithName(SelectorList list, KatjaString string) {
196         SelectorListIterator it = list.iterator();
197         while (it.hasNext()) {
198             Selector sel = it.next();
199             if (sel.selector().eq(string))
200                 return sel;
201         }
202         return null;
203     }
204
205     /**
206      * Returns 'KatjaElement'
207      * @see katja.ClassGenerator#getExtendedClass()
208      */
209     protected String getExtendedClass() {
210         return "KatjaElement";
211     }
212
213     protected boolean isInterface() {
214         return true;
215     }
216
217     protected List getSortIdNames() {
218         LinkedList list = new LinkedList();
219
```

```
220         // First add all children
221         SortIdListIterator it = children.iterator();
222         while (it.hasNext()) {
223             list.add(it.next().name().toString());
224         }
225
226         // Then add all return types of selector methods
227         SelectorListIterator it2 = getAllSelectors().iterator();
228         while (it2.hasNext()) {
229             list.add(it2.next().sortId().name().toString());
230         }
231
232         return list;
233     }
234
235     protected Collection getImportList() {
236         Collection col = super.getImportList();
237         col.add(new String("katja.common.*"));
238         return col;
239     }
240 }
```

ListGenerator.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7
8  import java.io.IOException;
9  import java.util.Collection;
10 import java.util.LinkedList;
11 import java.util.List;
12
13 import katja.spec.ListProd;
14
15 /**
16  * Class to generate List classes
17  *
18  * @author Jan Sch&auml;fer
19  */
20 public class ListGenerator extends ClassGenerator {
21     String childType;
22     ListIteratorGenerator iteratorGen;
23
24     /**
25      * Creates a new KatjaListGenerator
26      * @param pkgGen the PackageGenerator
27      * @param prod the ListProd to create the class from
28      */
29     public ListGenerator(
30                     PackageGenerator pkgGen,
31                     ListProd prod)
32     {
33         super(pkgGen, prod);
34         this.childType = prod.param().name().stringValue();
35         iteratorGen = new ListIteratorGenerator(pkgGen,prod);
36     }
```

```
37
38      public void generate() throws IOException {
39          super.generate();
40          iteratorGen.generate();
41      }
42
43
44      /**
45       * Generates nothing, as the member variables are defined
46       * in KatjaListImpl
47       * @see katja.classGenerator#generateMemberVariables()
48       */
49      protected void generateMemberVariables() {
50      }
51
52      protected void generateMethods() {
53          generateSubterm();
54          generateGet();
55          generateContains();
56          generateFirst();
57          generateLast();
58          generateAppFront();
59          generateAppBack();
60          generateAdd();
61          generateRemove();
62          generateRemoveAll();
63          generateFront();
64          generateBack();
65          generateConc();
66          generateAddAll();
67          generateIterator();
68          generateCreateInstance();
69      }
70
71      private void generateGet() {
72          writer.println("    public "+childType+" get(int ith) {");
73          writer.println("        return ("+childType+") subtermInternal(ith);");
74          writer.println("    }");
75          writer.println();
76      }
77
78      private void generateContains() {
79          writer.println("    public boolean contains("+childType+" e) {");
80          writer.println("        return containsInternal(e);");
81          writer.println("    }");
82          writer.println();
83      }
84
85      private void generateRemove() {
86          writer.println("    public "+className+" remove("+childType+" e) {");
87          writer.println("        return ("+className+") removeInternal(e);");
88          writer.println("    }");
89          writer.println();
90      }
91
92      private void generateRemoveAll() {
93          writer.println("    public "+className+" removeAll("+className+" e) {");
94          writer.println("        return ("+className+") removeAllInternal(e);");
95          writer.println("    }");
96          writer.println();
97      }
98
```

```
99
100     private void generateAdd() {
101         writer.println("    public "+className+" add("+childType+" e) {");
102         writer.println("        return ("+className+") appBackInternal(e);");
103         writer.println("    }");
104         writer.println();
105     }
106
107     private void generateAddAll() {
108         writer.println("    public "+className+" addAll("+className+" l) {");
109         writer.println("        return ("+className+") concInternal(l);");
110         writer.println("    }");
111         writer.println();
112     }
113
114     private void generateSubterm() {
115         writer.println("    public "+childType+" subterm(int ith) {");
116         writer.println("        return ("+childType+") subtermInternal(ith);");
117         writer.println("    }");
118         writer.println();
119     }
120
121     private void generateFirst() {
122         writer.println("    public "+childType+" first() {");
123         writer.println("        return ("+childType+") firstInternal();");
124         writer.println("    }");
125         writer.println();
126     }
127
128     private void generateLast() {
129         writer.println("    public "+childType+" last() {");
130         writer.println("        return ("+childType+") lastInternal();");
131         writer.println("    }");
132         writer.println();
133     }
134
135     private void generateAppFront() {
136         writer.println("    public "+className+" appFront("+childType+" e) {");
137         writer.println("        return ("+className+") appFrontInternal(e);");
138         writer.println("    }");
139         writer.println();
140     }
141
142     private void generateAppBack() {
143         writer.println("    public "+className+" appBack("+childType+" e) {");
144         writer.println("        return ("+className+") appBackInternal(e);");
145         writer.println("    }");
146         writer.println();
147     }
148
149     private void generateFront() {
150         writer.println("    public "+className+" front() {");
151         writer.println("        return ("+className+") frontInternal();");
152         writer.println("    }");
153         writer.println();
154     }
155
156     private void generateBack() {
157         writer.println("    public "+className+" back() {");
158         writer.println("        return ("+className+") backInternal();");
159         writer.println("    }");
160         writer.println();
```

```
161     }
162
163     private void generateConc() {
164         writer.println("    public "+className+" conc("+className+" l) {");
165         writer.println("        return ("+className+") concInternal(l);");
166         writer.println("    }");
167         writer.println();
168     }
169
170     private void generateIterator() {
171         writer.println("    public "+className+"Iterator iterator() {");
172         writer.println("        return new "+className+"Iterator(values.iterator
                ());");
173         writer.println("    }");
174         writer.println();
175     }
176
177     private void generateCreateInstance() {
178         writer.println("    protected KatjaListImpl createInstance(List
                listValues) {");
179         writer.println("        "+className+" l = new "+className+"();");
180         writer.println("        l.values = listValues;");
181         writer.println("        return l;");
182         writer.println("    }");
183         writer.println();
184     }
185
186     /**
187      * Returns 'KatjaListImpl'
188      * @see katja.KatjaClassGenerator#getExtendedClass()
189      */
190     protected String getExtendedClass() {
191         return "KatjaListImpl";
192     }
193
194     protected Collection getImportList() {
195         Collection col = super.getImportList();
196         col.add(new String("katja.common.*"));
197         col.add(new String("java.util.List"));
198         return col;
199     }
200
201     protected List getSortIdNames() {
202         LinkedList list = new LinkedList();
203         list.add(childType);
204         return list;
205     }
206 }
```

ListIteratorGenerator.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7
8  import java.util.Collection;
9  import java.util.LinkedList;
```

```
10 import java.util.List;
11
12 import katja.spec.ListProd;
13
14 /**
15  * Class to generate ListIterator classes
16  * @author Jan Sch&auml;fer
17  */
18 public class ListIteratorGenerator extends ClassGenerator {
19
20     ListProd listProd;
21
22     public ListIteratorGenerator(
23         PackageGenerator katjaPackageGenerator,
24         ListProd prod) {
25         super(katjaPackageGenerator,prod.sortId().name()+"Iterator");
26         listProd = prod;
27     }
28
29     protected void generateMemberVariables() {
30         writer.println("    private Iterator iterator;");
31         writer.println();
32     }
33
34     protected void generateMethods() {
35         generateConstructor();
36         generateHasNext();
37         generateNext();
38     }
39
40     private void generateConstructor() {
41         writer.println("    public "+className+
42             "(Iterator it) {");
43         writer.println("        this.iterator = it;");
44         writer.println("    }");
45         writer.println();
46     }
47
48     private void generateHasNext() {
49         writer.println("    public boolean hasNext() {");
50         writer.println("        return iterator.hasNext();");
51         writer.println("    }");
52         writer.println();
53     }
54
55     private void generateNext() {
56         writer.println("    public "+listProd.param().name()+" next() {");
57         writer.println("        return ("+listProd.param().name()+") iterator.
               next();");
58         writer.println("    }");
59         writer.println();
60     }
61
62     /**
63      * returns <code>null</code>
64      * @return <code>null</code>
65      */
66     protected String getExtendedClass() {
67         return null;
68     }
69
70     /**
```

```
71          * Does nothing, as ListIterators have no sort
72          */
73         protected void generateSortDefinition() {
74         }
75
76         /**
77          * We don't want any sort code here, so
78          * we override the method and do nothing
79          */
80         protected void generateSortMethod() {
81         }
82
83         protected Collection getImportList() {
84             Collection col = super.getImportList();
85             col.add("java.util.Iterator");
86             return col;
87         }
88
89         protected List getSortIdNames() {
90             LinkedList list = new LinkedList();
91             list.add(listProd.param().name().toString());
92             return list;
93         }
94 }
```

Analyser.java

```
1  /*
2   * Copyright 2003
3   * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4   * All rights reserved.
5   */
6  package katja;
7
8  import java.util.HashMap;
9  import java.util.HashSet;
10 import java.util.Iterator;
11
12 import katja.common.KatjaInt;
13 import katja.common.KatjaString;
14 import katja.spec.*;
15
16 /**
17  * The Analyser
18  */
19 public class Analyser {
20
21     private Katja katja;
22
23     private int num_errors = 0;
24
25     /**
26      * Stores all defined SortIds
27      * Defined is a sortid after it has
28      * been declared as a tuple, list or variant
29      */
30     private HashSet definedSortIdHash;
31
32     /**
33      * Stores for every SortId the
```

```
34          * package where it is defined
35          */
36         private HashMap sortIdPackages;
37
38         /**
39          * Contains all Strings of names that are defined by
40          * java import statements.
41          */
42         private HashMap definedJavaImportHash;
43
44         public Analyser(Katja katja) {
45             this.katja=katja;
46             definedSortIdHash = new HashSet();
47             definedJavaImportHash = katja.javaImportHash;
48             sortIdPackages = katja.sortIdPackageHash;
49         }
50
51         public Analyser(Katja katja, Analyser parent) {
52             this.katja=katja;
53             this.definedJavaImportHash = parent.definedJavaImportHash;
54             this.definedSortIdHash = parent.definedSortIdHash;
55             this.sortIdPackages = parent.sortIdPackages;
56         }
57
58         public void analyse() {
59             analyseImports();
60             fillDefinedHashes();
61             checkDefinitionOfSortIds();
62         }
63
64         private void checkDefinitionOfSortIds() {
65             Specification spec = katja.getSpecification();
66             ProductionListIterator it = spec.prodList().iterator();
67             while (it.hasNext()) {
68                 Production prod = it.next();
69                 switch (prod.sort().toInt()) {
70                     case TupleProd.sortInt:
71                         checkTupleDefinition((TupleProd) prod);
72                         break;
73                     case ListProd.sortInt:
74                         checkListDefinition((ListProd) prod);
75                         break;
76                     case VariantProd.sortInt:
77                         checkVariantDefinition((VariantProd) prod);
78                         checkCyclicFreeness((VariantProd) prod);
79
80                         break;
81                 }
82             }
83         }
84
85
86         /**
87          * Checks if the variant is cyclicly defined<br>
88          * E.g. A = A would be a cycle, or A = B B = A, or
89          * A = B B = C C = A
90          *
91          * @param prod the variant to check
92          */
93         private void checkCyclicFreeness(VariantProd prod) {
94             findSortIdInVariants(prod.sortId(),prod);
95         }
```

```
96
97      private void findSortIdInVariants(SortId variant, VariantProd prod) {
98          SortIdList list = katja.getVariantsOfSortId(variant);
99          if (list == null)
100             return;
101
102         SortIdListIterator it = list.iterator();
103
104         while (it.hasNext()) {
105             SortId sortId = it.next();
106             if (sortId.eq(prod.sortId())) {
107                 error("Cyclic dependency of variant "+prod.sortId().name()+"
                detected!",prod.sortId().line());
108                 return;
109             } else {
110                 findSortIdInVariants(sortId, prod);
111             }
112         }
113     }
114
115     private void checkVariantDefinition(VariantProd prod) {
116         SortIdListIterator it = prod.params().iterator();
117
118         while (it.hasNext()) {
119             SortId id = it.next();
120             if (! sortIdIsDefined(id)) {
121                 error("Sort "+id.name()+" is undefined!",id.line());
122             } else
123             if (definedJavaImportHash.containsKey(id.name().toString())) {
124                 error("The variant '"+prod.sortId().name()+"' contains the Java
                type '"+id.name()+"', which is not allowed !",prod.sortId().line());
125             } else
126             if (Katja.isBuiltInType(id.name().toString())) {
127                 error("The variant '"+prod.sortId().name()+"' contains the built-
                in type '"+id.name()+"', which is not allowed !",prod.sortId().line()
                );
128             }
129
130         }
131     }
132
133
134     private void checkListDefinition(ListProd prod) {
135         if ( ! sortIdIsDefined(prod.param()))
136             error("Sort "+prod.param().name()+" is undefined!",prod.sortId().line
                ());
137     }
138
139     private void checkTupleDefinition(TupleProd prod) {
140         TupleParamListIterator it = prod.params().iterator();
141
142         while (it.hasNext()) {
143             TupleParam param = it.next();
144             SortId sortId;
145             if (param.is(SortId.sort))
146                 sortId = (SortId) param;
147             else
148                 sortId = ((Selector) param).sortId();
149
150             if (! sortIdIsDefined(sortId)) {
151                 error("Sort "+sortId.name()+" is undefined!",sortId.line());
152             }
```

```
153         }
154     }
155
156     private boolean sortIdIsDefined(SortId id) {
157         if (Katja.isBuiltInType(id.name().toString()))
158             return true;
159
160         if (definedJavaImportHash.containsKey(id.name().toString()))
161             return true;
162
163         if (definedSortIdHash.contains(id.name()))
164             return true;
165
166         if (isInClassPath(id.name().toString()))
167             return true;
168
169         return false;
170     }
171
172     private boolean isInClassPath(String string) {
173         if (findClass(string))
174             return true;
175
176         if (findClass("java.lang."+string))
177             return true;
178
179         return false;
180     }
181
182     private boolean findClass(String string) {
183         try {
184             this.getClass().getClassLoader().loadClass(string);
185             Katja.debug("Sort "+string+" is in classpath!");
186             return true;
187         }
188         catch (ClassNotFoundException e) {
189             Katja.debug("Sort "+string+" is in NOT classpath!");
190             return false;
191         }
192     }
193
194     private void analyseImports() {
195         Iterator it = katja.getKatjaImports();
196         while (it.hasNext()) {
197             Katja m = (Katja) it.next();
198             Analyser analyser = new Analyser(m,this);
199             analyser.analyse();
200             num_errors += analyser.num_errors;
201         }
202     }
203
204     private void fillDefinedHashes() {
205         fillDefinedJavaImportHash(katja.getSpecification());
206         fillDefinedSortIdHash(katja.getSpecification());
207     }
208
209     private void fillDefinedJavaImportHash(Specification spec) {
210         ImportListIterator it = spec.importList().iterator();
211         while (it.hasNext()) {
212             Import imp = it.next();
213             if (imp.is(JavaImport.sort)) {
214                 if (imp.is(JavaSimpleImport.sort)) {
```

```
215                     String name = getNameOfImport(imp);
216                     definedJavaImportHash.put(name,imp);
217                     Katja.debug("Adding Java type : "+name);
218                 }
219                 else
220                     error("Java imports on demand aren't supported yet!",imp.line
                ());
221                 }
222             }
223         }
224 
225         /**
226          * Returns the unqualified classname of the Import.<br>
227          * E.g. for the import 'java.util.Iterator' it would return
228          * 'Iterator'
229          * @param imp the Import to return the classname of
230          * @return the classname of the Import
231          */
232         private String getNameOfImport(Import imp) {
233             String name = imp.name().toString();
234             int dot = name.lastIndexOf(".");
235             name = name.substring(dot+1,name.length());
236             return name;
237         }
238 
239         private void fillDefinedSortIdHash(Specification spec) {
240             ProductionListIterator it = spec.prodList().iterator();
241             while (it.hasNext()) {
242                 Production prod = it.next();
243                 if (definedSortIdHash.contains(prod.sortId().name())) {
244                     error("Sort "+prod.sortId().name()+" is already defined !",prod.
                sortId().line());
245                 } else {
246                     definedSortIdHash.add(prod.sortId().name());
247                     PackageDcl pck = spec.pck();
248                     if (pck.is(Empty.sort)) {
249                         sortIdPackages.put(prod.sortId().name(), "");
250                         Katja.debug("Adding empty packagename to sort");
251                     }
252                     else {
253                         PackageName pckName = (PackageName) pck;
254                         sortIdPackages.put(prod.sortId().name().toString(), pckName.
                name().toString());
255                         Katja.debug("Adding packagename to sort: "+pckName.name());
256                     }
257 
258                     Katja.debug("Adding defined sort: "+prod.sortId().name());
259                 }
260             }
261         }
262 
263         private void error(String string,KatjaInt line) {
264             num_errors++;
265             katja.addError(new KatjaError(
266                 new KatjaString(string),
267                 new KatjaString(katja.getSpecFile()),
268                 line));
269         }
270 
271         public int numberOfErrors() {
272             return num_errors;
273         }
```

```
274
275     public HashMap getJavaImportHash() {
276         return definedJavaImportHash;
277     }
278 }
```

KatjaElement.java

```
1  package katja.common;
2
3
4  public interface KatjaElement {
5      public final static int sortInt = 1;
6      public final static KatjaSort sort = new KatjaSort(KatjaElement.class,sortInt
               );
7
8      public KatjaSort sort();
9      public boolean is(KatjaSort s);
10     public boolean eq(KatjaElement e);
11     public boolean equals(Object o);
12     public int hashCode();
13     public String toString();
14 }
```

KatjaSort.java

```
1  package katja.common;
2
3  public final class KatjaSort extends KatjaElementImpl {
4
5      public final static int sortInt = 5;
6      public final static KatjaSort sort = new KatjaSort(KatjaSort.class,5);
7
8      private Class cl;
9      private int id;
10
11     protected Class getClassValue() {
12         return cl;
13     }
14
15     public KatjaSort(Class cl,int id) {
16         this.cl = cl;
17         this.id = id;
18     }
19
20     public KatjaSort sort() {
21         return KatjaSort.sort;
22     }
23
24     public boolean eq(KatjaElement s)  {
25         if (!(s instanceof KatjaSort))
26             return false;
27
28         return this.id == ((KatjaSort)s).id;
29     }
30
31     public String toString() {
32         return cl.getName();
```

```
33      }
34
35      public int toInt()  {
36          return id;
37      }
38
39      public int intValue() {
40          return id;
41      }
42
43      public int hashCode() {
44          return id;
45      }
46
47  }
```

KatjaTerm.java

```
1   package katja.common;
2
3
4   public interface KatjaTerm extends KatjaElement {
5       public final static int sortInt = 2;
6       public final static KatjaSort sort = new KatjaSort(KatjaTerm.class,sortInt);
7
8       public int numSubterms();
9       public int size();
10  }
```

KatjaBuiltInType.java

```
1   package katja.common;
2
3   public abstract class KatjaBuiltInType extends KatjaElementImpl implements
                KatjaTerm {
4       public final static int sortInt = 6;
5       public final static KatjaSort sort = new KatjaSort(KatjaBuiltInType.class,6);
6
7       public int numSubterms() {
8           return 0;
9       }
10
11      public int size() {
12          return 0;
13      }
14  }
```

KatjaTuple.java

```
1   package katja.common;
2
3
4   public interface KatjaTuple extends KatjaTerm {
5       public final static int sortInt = 3;
6       public final static KatjaSort sort = new KatjaSort(KatjaList.class,3);
7
```

```
8       public Object subterm(int ith);
9   }
```

KatjaList.java

```
1   package katja.common;
2
3
4   public interface KatjaList extends KatjaTerm{
5       public final static int sortInt = 4;
6       public final static KatjaSort sort = new KatjaSort(KatjaList.class,4);
7
8   }
```

KatjaElementImpl.java

```
1   package katja.common;
2
3   public abstract class KatjaElementImpl implements KatjaElement {
4       public boolean is(KatjaSort s)  {
5           return s.getClassValue().isInstance(this);
6       }
7
8       public boolean equals(Object o) {
9           if ( ! (o instanceof KatjaElement))
10              return false;
11
12          return eq((KatjaElement) o);
13      }
14  }
```

KatjaTermImpl.java

```
1   package katja.common;
2
3   public abstract class KatjaTermImpl extends KatjaElementImpl implements KatjaTerm
                {
4       public int size() {
5           return numSubterms();
6       }
7   }
```

KatjaTupleImpl.java

```
1   package katja.common;
2
3   public abstract class KatjaTupleImpl extends KatjaTermImpl implements KatjaTuple
                {
4   }
```

KatjaListImpl.java

```
1   package katja.common;
```

```
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.List;
6
7  public abstract class KatjaListImpl extends KatjaTermImpl implements KatjaTerm {
8      protected List values = new ArrayList();
9
10     protected abstract KatjaListImpl createInstance(List listValues);
11
12     protected Object subtermInternal(int ith)  {
13         if (ith < 0)
14             throw new KatjaIllegalArgumentException("Trying to access subterm "+
15                     ith+", but only values greater 0 are allowed !");
16
17         if (ith >= values.size())
18             return null;
19
20         return values.get(ith);
21     }
22
23     public int numSubterms()  {
24         return values.size();
25     }
26
27     public int size() {
28         return values.size();
29     }
30
31     protected boolean containsInternal(Object o) {
32         return values.contains(o);
33     }
34
35     protected Object firstInternal()  {
36         if (values.size() == 0)
37             return null;
38
39         return values.get(0);
40     }
41
42     protected Object lastInternal()  {
43         if (values.size() == 0)
44             return null;
45
46         return values.get(values.size());
47     }
48
49     protected KatjaListImpl frontInternal()  {
50         if (values.size() == 0)
51             return null;
52
53         return createInstance(values.subList(0, values.size()-1));
54    }
55
56     protected KatjaListImpl backInternal()  {
57         if (values.size() == 0)
58             return null;
59
60         return createInstance(values.subList(1, values.size()));
61      }
62
63     protected KatjaListImpl appFrontInternal(Object o)  {
```

```
64          List list = new ArrayList(values.size()+1);
65          list.add(o);
66          list.addAll(values);
67
68          return createInstance(list);
69      }
70
71      protected KatjaListImpl removeInternal(Object o)  {
72          if (values.size() == 0)
73              return null;
74
75          List list = new ArrayList(values);
76          list.remove(o);
77
78          return createInstance(list);
79      }
80
81      protected KatjaListImpl removeAllInternal(KatjaListImpl l)  {
82          if (values.size() == 0)
83              return null;
84
85          List list = new ArrayList(values);
86          list.removeAll(l.values);
87
88          return createInstance(list);
89      }
90
91
92      protected KatjaListImpl appBackInternal(Object o)  {
93           List list = new ArrayList(values.size()+1);
94           list.addAll(values);
95           list.add(o);
96
97           return createInstance(list);
98      }
99
100     protected KatjaListImpl concInternal(KatjaListImpl l)  {
101         List list = new ArrayList(values.size()+l.values.size());
102         list.addAll(values);
103         list.addAll(l.values);
104
105         return createInstance(list);
106     }
107
108     public boolean eq(KatjaElement e)  {
109         if (! (e instanceof KatjaListImpl))
110                 return false;
111
112         KatjaListImpl el = (KatjaListImpl) e;
113         return el.values.equals(values);
114     }
115
116     public int hashCode() {
117         return values.hashCode()+this.getClass().hashCode()*31;
118     }
119
120     public String toString() {
121         Iterator it = values.iterator();
122         StringBuffer result = new StringBuffer();
123         result.append("[ ");
124         while (it.hasNext()) {
125             result.append(it.next().toString());
```

```
126             if (it.hasNext())
127                 result.append(", ");
128         }
129         result.append(" ]");
130 
131         return result.toString();
132     }
133 }
```

KatjaString.java

```
1  package katja.common;
2  
3  
4  public class KatjaString extends KatjaBuiltInType {
5      public final static int sortInt = 9;
6      public final static KatjaSort sort = new KatjaSort(KatjaString.class,sortInt)
           ;
7  
8      String value;
9  
10     public KatjaString(String value) {
11         this.value = value;
12     }
13 
14     public String toString() {
15         return value;
16     }
17 
18     public String stringValue() {
19         return value;
20     }
21 
22     public KatjaSort sort() {
23         return KatjaString.sort;
24     }
25 
26     public boolean eq(KatjaElement e)  {
27         if (!(e instanceof KatjaString))
28             return false;
29 
30         return this.value.equals(((KatjaString)e).value);
31     }
32 
33     public int hashCode() {
34         return value.hashCode();
35     }
36 }
```

KatjaInt.java

```
1  package katja.common;
2  
3  
4  public class KatjaInt extends KatjaBuiltInType {
5      public final static int sortInt = 8;
6      public final static KatjaSort sort = new KatjaSort(KatjaInt.class,sortInt);
7  
```

```
8       int value;
9
10      public KatjaInt( int value) {
11          this.value = value;
12      }
13
14      public int toInt()  {
15          return value;
16      }
17
18      public int intValue() {
19          return value;
20      }
21
22      public KatjaSort sort() {
23          return KatjaInt.sort;
24      }
25
26      public boolean eq(KatjaElement e)  {
27          if (! (e instanceof KatjaInt))
28              return false;
29
30          return ((KatjaInt)e).toInt() == this.toInt();
31      }
32
33      public int hashCode() {
34          return value;
35      }
36
37      public String toString() {
38          return Integer.toString(value);
39      }
40
41  }
```

KatjaChar.java

```
1   package katja.common;
2
3
4   public class KatjaChar extends KatjaBuiltInType {
5       public final static int sortInt = 7;
6       public final static KatjaSort sort = new KatjaSort(KatjaChar.class,sortInt);
7
8       char value;
9
10      public KatjaChar(char value) {
11          this.value = value;
12      }
13
14      public char toChar()  {
15          return value;
16      }
17
18      public char charValue() {
19          return value;
20      }
21
22      public KatjaSort sort() {
23          return KatjaChar.sort;
```

```
24      }
25
26      public boolean eq(KatjaElement e)  {
27          if (! (e instanceof KatjaChar))
28              return false;
29
30          return ((KatjaChar)e).value == this.value;
31      }
32
33      public int hashCode() {
34          return value;
35      }
36
37      public String toString() {
38          return Character.toString(value);
39      }
40  }
```

KatjaBool.java

```
1   package katja.common;
2
3
4   public class KatjaBool extends KatjaBuiltInType {
5       public final static int sortInt = 10;
6       public final static KatjaSort sort = new KatjaSort(KatjaBool.class,10);
7
8       public static final KatjaBool maxTrue = new KatjaBool(true);
9       public static final KatjaBool maxFalse = new KatjaBool(false);
10
11      boolean value;
12
13      private KatjaBool(boolean value) {
14          this.value = value;
15      }
16
17      public static KatjaBool fromBoolean(boolean b) {
18          return b ? maxTrue : maxFalse;
19      }
20
21      public boolean toBoolean()  {
22          return value;
23      }
24
25      public boolean booleanValue() {
26          return value;
27      }
28
29      public boolean eq( KatjaElement e)  {
30          return this == e;
31      }
32
33      public KatjaSort sort() {
34          return KatjaBool.sort;
35      }
36
37
38      public int hashCode() {
39          return value ? 5 : 31;
40      }
```

```
41
42      public String toString() {
43          return value ? "true" : "false";
44      }
45
46  }
```

KatjaUnkownTypeException.java

```
1   /*
2    * Copyright 2003
3    * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4    * All rights reserved.
5    */
6   package katja.common;
7
8   public class KatjaUnkownTypeException extends KatjaException {
9
10      public KatjaUnkownTypeException(int typeConstant) {
11          super("Unkown type : "+typeConstant);
12      }
13
14  }
```

KatjaIllegalMethodException.java

```
1   /*
2    * Copyright 2003, AG Software Technology , University of Kaiserslautern
3    * Created on 08.05.2003
4    *
5    */
6   package katja.common;
7
8   /**
9    *
10   * @author Jan Sch&auml;fer
11   */
12  public class KatjaIllegalMethodException extends KatjaException {
13      public KatjaIllegalMethodException(String msg) {
14          super(msg);
15      }
16  }
```

KatjaIllegalArgumentException.java

```
1   /*
2    * Copyright 2003
3    * AG Softwaretechnik, Universitaet Kaiserslautern, Germany
4    * All rights reserved.
5    */
6   package katja.common;
7
8   public class KatjaIllegalArgumentException extends KatjaException {
9       public KatjaIllegalArgumentException(String msg) {
10          super(msg);
11      }
```

```
12 }
```

KatjaException.java

```
1  /*
2   * Copyright 2003, AG Software Technology , University of Kaiserslautern
3   * Created on 08.05.2003
4   *
5   */
6  package katja.common;
7
8  /**
9   *
10  * @author Jan Sch&auml;fer
11  */
12 public class KatjaException extends RuntimeException {
13      public KatjaException(String msg) {
14          super(msg);
15      }
16 }
```