Abaco Quick User’s Guide

Luis Carlos de Sousa Menezes

1. Introduction

This is a quick introduction for the Abaco System. This system is intended to be an action semantics based compiler generation system that is able to recognize action semantics descriptions and produce implementations that can be used to test the language’s features before to produce the real implementations (we expect that Abaco can produce these final implementations some day).

The next sections will describe concepts for the Graphical User Interface, the Algebraic Specification Language, the Action Semantics Support and how to integrate the System with Java Code.

2. Graphical User Interface

This section explains some basic concepts used by Abaco Graphical User Interface.

2.1 Projects

The Abaco Interface is designed work with project files. A project file contains a list of specifications that can be compiled by the Abaco Compiler.

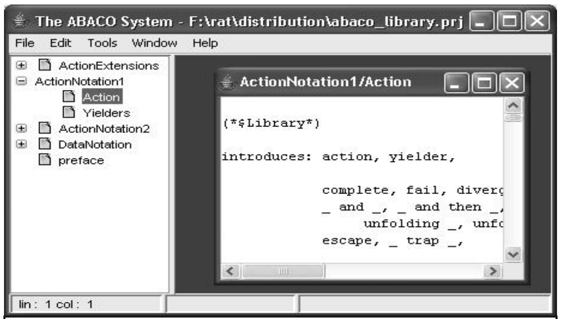
By default, the Abaco is initialized with no project currently opened. Before we can edit specifications we have to create a new project (Menu Option “**File/New/Project**”) or open an existing one (Menu Option “**File/Open**”). Once the user chooses a project he is able to edit or compile its specifications.

When the user finishes to edit one project he can save the modified project (Menu Options “**File/Save**” or “**File/Save as...**”), close the current project (Menu Option “**File/Close**”) or exit the system.

2.2 Main Window

The Abaco main interface, showed by the Figure 1, is formed by two work areas: Specifications panel (the left side panel) and the editors panel (the right one).

The specifications panel displays the existing specifications in the currently opened project using a hierarchical tree (the specification “**ActionNotation/Functional**” is showed like a node “**Functional**” child of the node “**ActionNotation**”). Some nodes of the project tree may contain no associated specification (the project contains a specification named “**ActionNotation/Functional**” but no specification named “**ActionNotation**”). This feature is modeled by the icon (🗎), placed beside the specification name, that indicates the existence of one specification associated with this node.



```
(*$Library*)

introduces: action, yielder,

            complete, fail, diverg
            _ and _, _ and then _,
                unfolding _, unfo
            escape, _ trap _,
```

Figure 1Abaco User Inteface

The editors panel are used to hold several kinds of editors to allow the user to change the specifications' contents, evaluate terms in a compiled specifications, type actions to be executed, etc.

2.3 Editing Projects

Once a project is opened, the user can execute the following editing actions in the system:

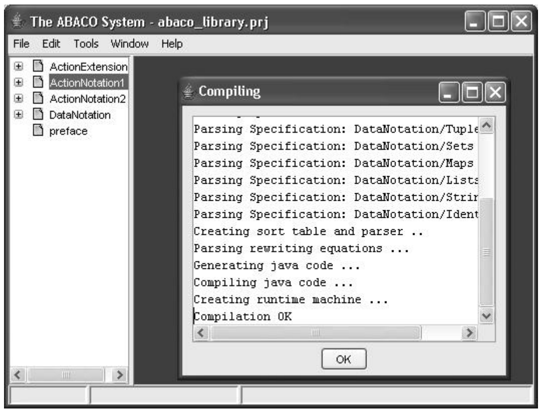
Editing a specification: If the user wants to change the text associated with a specification, he just have to make a double-click in a specification panel item. This action will open a specification editor window in editors panel with the specifications contents. Changes in this window will alter the associated specification.

Create a new specification: A new specification can be inserted selecting the menu option ("**File/New/Specification**") or doing a right-click in specifications panel and select the "**New Specification**" options. The System will ask the specification's name and create a new node in specifications panel and open a specification editor window.

Remove a specification: The Abaco has two kinds of specification remove operation: a recursive and a non-recursive one. The first operation will delete the selected specification and also its child specifications. The second one just removes the selected specification.

2.4 Compiling Specifications

To compile one specification in Abaco, initially select a specification in the specifications panel. After you select the specification click the right mouse button to shows the menu for this specification. Select the option compile to start the compilation process. The Abaco will show a compiler window that indicates the compilation's current stage. If some error is found the system will abort the compilation indication the error found. If the process finishes successfully you should get a screen like the following figure:

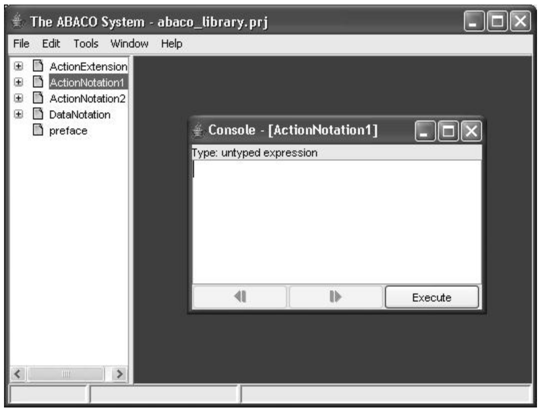


```
Parsing Specification: DataNotation/Tuple
Parsing Specification: DataNotation/Sets
Parsing Specification: DataNotation/Maps
Parsing Specification: DataNotation/Lists
Parsing Specification: DataNotation/Strir
Parsing Specification: DataNotation/Ident
Creating sort table and parser ..
Parsing rewriting equations ...
Generating java code ...
Compiling java code ...
Creating runtime machine ...
Compilation OK
```

Select “OK” and Abaco will open a console window that allows the evaluation of valid terms according the specification’s rules.

2.5 Executing Compiled Specifications

If a specification was successfully compiled, the Abaco opens a console window that allows the evaluation of valid terms like defined in compiled specification. The following figure shows an interface with this kind of window:



The console window is formed by: a **text area** that is used to edit term to be evaluated, two **navigation buttons** that navigate among past evaluated terms an the **execution button** that evaluates the existing term and shows its value after the specification’s equations processing.

2.6 Saving the Compiled Code

If you are satisfied with the compiled code you can save it to use outside the environment. To save the binary code select the console window and execute the menu option (“**Tools/Save Generated Code ...”**) one is displayed asking for the saved code filename.

2.7 Using the Saved Code

The produced binary code can be used as shown bellow:

a) As a java standalone application

Just executes in a command line: “java –jar SavedCode.jar” and the system will popup an evaluation window similar to the Environment Console window. The archive: “**runtime.jar**”, that contains Abaco code used by compiled specification should be located in the same place of “**SavedCode.jar**” or pointed by the CLASSPATH environment variable.

b) As an Applet

It is possible to use a save code as an applet inserting in a HTML file the following code segment:

```
<applet code=GuiApplet archive="SavedCode.jar,runtime.jar" WIDTH=600 HEIGHT=400 >
  <param name="label1" value="Evaluate" >
  <param name="label2" value="Compile" > <param name="action2" value="compile _" >
  <param name="contents" value="give 0" >
  <param name=”contentsURL” value=”teste.an" >
</applet>
```

This code segment when interpreted by a browser will show a console like window inside the HTML page. The console window can be action's buttons configured by the pair of arguments (label$_x$, action$_x$), where label$_x$ indicates the button's label and action$_x$ indicates an specification's operator that will process the term typed by user, if no action is associated with a label it will produce a button that just evaluate the typed term according the compiled specification's rules.

The "**contents**" and "**contentsURL**" arguments can be used to specify the text box's initial value.

OBS.: In the present system version the applet code presented some instantiation problems in some older versions of the Internet explorer and Netscape or newer browsers with an old version of the Java virtual machine (they can't be compatible with the binary code versions produced).

c) **Embedded in other applications**

It's possible to use Abaco's generated code inside applications code using the communication API. This API is described in Section .The code that makes this link is something like:

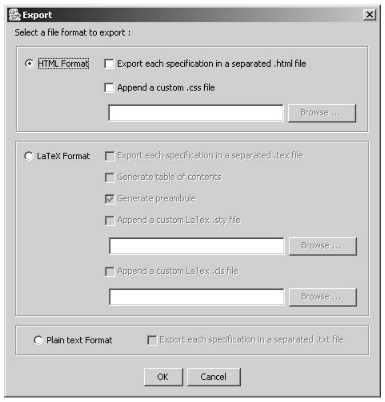
```
import br.ufpe.abaco.runtime.*;

....

AbacoExecutionEngine engine =
     AbacoExecutionEngine.fromJarFile(
         (new File("code.jar")).toURL()
     );
Term t = engine.parse("fib s s s 0");
System.out.println(engine.print(t));
```

2.8 Exporting Specification

The Abaco can export specification into text , LaTeX and HTML files. To export some specification, the user just have to select the desired main specification, click the right mouse button and select the option "**Export**". The Abaco should display an options window like the showed bellow:



Export

Select a file format to export :

[✓ HTML Format] Export each specification in a separated .html file
Append a custom .css file
Browse ...

LaTeX Format
Export each specification in a separated .tex file
Generate table of contents
[✓ Generate preambule]
Append a custom LaTex .sty file
Browse ...
Append a custom LaTex .cls file
Browse ...

Plain text Format
Export each specification in a separated .txt file

OK Cancel

These options help the configuration of the generated code. After this dialog, the system ask a filename and process the specifications to produce the exported document. All dependents specifications are also showed in the produced file.

3 Abaco Algebraic Specification Language

3.1 Preface

The Abaco Specification Language can be initiated by an introduces sentence and an needs sentences like we show in the following specification:

```
introduces: stack, Item.
needs: LinkedList.
```

The introduces sentence is used to defined witch sorts (data types) are defined by the specification. In the current Abaco version the user is not forced to declare explicitly all sorts defined by the specification (the system just produces a warning message if it happens).

The needs sentence specifies others specifications that, probably, contain definitions used by the current specification and should be imported by the compiler before process the specification. The abaco system by default searches the imported specifications inside the current project. If the current project does not contains the imported specification, the Abaco can also search in the project stored in the file "**abaco_library.prj**" this project may be located in the current directory or in the Abaco installation directory. The existence of this project is not obligatory and its missing will not produce any error to the compiler. If the imported specification were not found, the compiler aborts the process with the appropriated message. By default all Abaco specifications also imports a specification named "**preface**" that defines some basic characteristics used by the specifications (Pretty Printers operators for example).

3.2 Defining Sorts

The following kinds of definitions are used to define the relation between the sorts defined by specifications:

```
eq stack <= LinkedList .
eq number = integer.
```

The first definition specifies that **Stack** is a subsort of the sort **LinkedList**. This declaration makes with every term belonging to **stack** also belongs to **LinkedList**.

The second definition specifies that two sorts (**number** and **integer**) are equals and refers to the same data type.

3.3 Defining Operators

Operator functionality is used to define new operators signature. An operator is an entity that takes some arguments of defined types and produces a new valor of a determined type. The operator's definition syntax is shown bellow:

```
op push _ _ :: item, stack ->  stack.
op pop _ :: stack -> stack .
op empty-stack :: -> stack.
```

These sentences define four new operators in the specification. The first operator "**push _ _**" receives a item term and a stack and produces another stack term. In the operators name, the symbol "_" means the arguments placement. The second sentence is similar to the first one: it defines the operator "pop _" that receiver a stack element and produces another stack as answer. The last definition defines a constant operator (that receives no arguments) “**empty-stack**” of type “**stack”**.

3.3 Defining Rewriting Equations

Rewriting equations are used to specify operations semantics by defining the equality of different terms. The basic syntax of Abaco rewriting equations is exemplified below:

```
re pop push ~x ~y = ~y.
re sum ( 0 , ~x) = ~x.
```

3.5 BNF-Like Syntax Definitions

The Abaco has a alternative way to define operators that presents a syntax similar to BNF-Grammars and is good to define programming languages:

```
syntax
  Command =
     [[ “if” Expression “then” Command “else” Command ]] |
     [[ “while” Expression “do” Command ]] |
     [[ Command “;” Command ]] .
  Expression = [[ Expression “+” Expression ]] | [[ Number ]] .
endsyntax
```

This specification is similar to the following one:

```
op if _ then _ else _ :: Expression, Command, Command -> Command .
op while _ do _ :: Expression, Command -> Command.
op _ ; _ :: Command, Command -> Command.
op _ + _ :: Expression, Expression -> Expression.
eq Number <= Expression .
```

3.6 Lexical Rules

Lexical rules allows the definition of more complex sorts using regular expressions:

```
lexical [0-9]+ :: number .
lexical \”[a-z0-9A-Z ]*\” :: string.
```

The first sentence defines that a sequence of elements between “0” and “9” makes a constant of type “**number**”. The second one defines that a sequence of letters, number and white spaces delimited by double-quotes forms elements of type “**string**”.

3.7 Pretty Printing

Pretty Printer equations are used to specify how the Abaco should typeset specification's terms when the default way is not adequate (produces ugly results). The following code segment exemplifies this feature:

```
pp if _ then _ else _ ::
    "if" %0 "then" prefix ("   ", %1) "else" prefix( "   ", %2 ) .
pp _ ; _ :: %0 nl %1 .
```

A pretty-printer rule is defined an operator and the rule that typesets terms produced by this operator. The typesetting rule is formed by a sequence of the following command:

- "String" – prints the specified String;
- %n – prints the nth argument;
- nl – jumps to the next line
- prefix ("string" , rule): print rule in a separated line

3.8 Interface with Native Java Code

To provide a more efficient way to implement some low level features, we can implement some specification's entities using native java code, using the following directive.

```
lexical [0-9]+ :: number => "NaturalTerm".
native sum ( _ , _ ) :: "NaturalSum" .
op _ and _ :: action, action -> action. (implemented by "OptimizedTerm")
```

The first directive defines sort elements using regular expressions and specifies that terms produced with this rule will be implemented by the class "NaturalTerm" that may represent numbers using more efficient ways that the default one (strings). "NaturalTerm" should be a class that extends the class "Term" (see Section 4) and must have a static object creator function: "static Term createTerm(String t)". That will be used to create this class objects.

The second directive specifies that the class "NaturalSum" implements the operator "sum (_ , _)". This class should extend the "NativeCode" derived class appropriated to operator's arity (see Section 4).

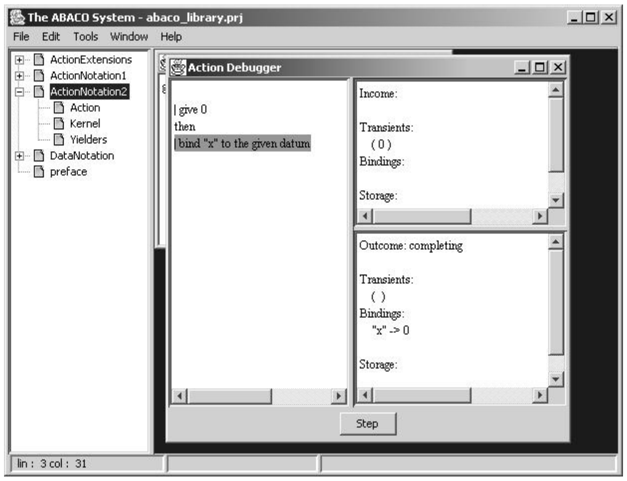
The last directive (not supported yet) extends the operator's functionality by defining a new class "OptimizedTerm" that will implement terms produced by this operator in a more efficient way than the default one.

4 The action semantics implementation

To facilitate the testing of action semantics descriptions, the Abaco system contains some facilities to help a designer to test specifications.

Action Notation and Data Notation Specifications: The "**abaco_library.prj**" project file contains specifications that define sorts and operators used by action semantics to describe programming languages semantics.

Action interpreter and debugger: the action interpreter and debugger allows designers to execute action produced by programming languages semantics and see their execution results in order to validate them. These tools can be called from console windows produced by the compilation of programming language specifications using the operators "**perform _**" and "**debug _**". These operators receive an action and, respectively, show the action execution result or open an action debugger window similar the showed in the next figure:



```
| give 0
then
| bind "x" to the given datum
```

```
Income:

Transients:
  ( 0 )
Bindings:

Storage:
```

```
Outcome: completing

Transients:
  ( )
Bindings:
  "x" -> 0

Storage:
```

The action debugger window contains panels that show, respectively, the action that is being executed, the informations passed to the action and their produced ones.

A generic (independently of programming language) action interpreter and debugger can be obtained by selecting the menu option "**File\New\Action Editor"** open an action editor window where actions can be typed and evaluated (menu option "**Tools\Interpret Action**") or a debugger window can be launched (menu option "**Tools\Debug Action"**)

5 *Interface with Java Code*

The Abaco systems defines some classes that can be used to call Abaco implementations from Java programs or implement some basic sorts using more efficient "low" level java constructions. These classes are located in the package "**br.ufpe.abaco.runtime**" which is stored in the runtime binary file of Abaco distribution "**runtime.jar**".

The first important class is the class Term that is parent of all classes that implements terms defined in compiled specifications. The term specification contains the functions showed bellow:

```
public abstract class Term {

Operator operator();

int arity();
Term getArg(int pos);

void setType(int t);
int getType();
}
```

The method **operator()** returns the specification's operator used to produce this term or **null** if it was not applicable.
The method **arity** return the number of arguments of this term. The method **getArg** is used to retrieve the nth argument for this term or null if not applicable.
The methods **setType** and **getType** are used to set the term type (sort).

The next class used by the Abaco's interface is the class **AbacoExecutionEngine** that models an compiled java specifications. The most important method's signatures of this class is showed bellow:

```
class AbacoExecutionEngine {

   Term parse(InputStream source)
     throws CompilerException, IOException;
   Term parse(String contents)
     throws CompilerException;
   String print(Term t);

   Operator getOperator(String name);
   int sortIndex(String sortName);
   String sortName(int sortIndex);

   Term run(Operator op);
   Term run(Operator op, Term a0);
   Term run(Operator op, Term a0, Term a1);
   Term run(Operator op, Term a0, Term a1, Term a2);
   Term run(Operator op, Term args[]);

   boolean isSubSort(Term t, int sortIndex);

   static AbacoExecutionEngine fromJarFile(URL fileName)
}
```

The method **parse** is intended to produce a term formed by the compilation of a character sequence, given by a String or a generic InputStream, processed by specification's rewriting equations.
The method **print** produces a human-readable string representation for a given Term. This string can be obtained using the default process or pretty-printer rules when applicable.
The methods **getOperator**, **sortIndex** and **sortName** are intended to retrieve operators and sorts runtime identifications from their specification names.
The methods **run** is intended to produced a term formed by the result of some operator applied with appropriated argument terms.
The methods **isSubSort** tests the type of some term
Finaly, the static method **fromJarFile** is intended to produce an execution engine from a Jar file that contains an specification code generated by Abaco Interface.

The last class we discuss is the class NativeCode and their subclasses. These classes are designed to holds java code that executes natively specification's defined operators.

```
class NativeCode {
   public AbacoExecutionEngine runner;
   public void setEngine(AbacoExecutionEngine runner);
}

public abstract class NativeCode0 extends NativeCode {
    abstract Term run();
}
public abstract class NativeCode1 extends NativeCode {
    public abstract Term run(Term arg0);
}
public abstract class NativeCode2 extends NativeCode {
    public abstract Term run(Term arg0,Term arg1);
}
.....
```

The classes **NativeCode0, NativeCode1, NativeCode2, NativeCode3, etc.** are intended to model, respectively operators with arity 0,1,2,3,etc. It contains a field that refers the current execution engine and a method **run** that should be redefined by derived classes and will be called when the compiled code needs the results of some operator applied some arguments. The method **run** should returns the resulted term or **null** value if it was not able to provide any term.