UPV-Curry User's Manual

S. Escobar M. Alpuente S. Lucas
{sescobar,alpuente,slucas}@dsic.upv.es

28th December 1998

1 Introduction

Currently, declarative programming is split into two main fields, namely functional programming and logic programming. Functional programming principally provides sophisticated abstraction facilities, module systems and clean solutions for integrating I/O into declarative programming, whereas logic programming allows for computing with partial information and provides built-in search facilities which have strong applications for knowledge-based systems and operations research. Recently, the execution principles of functional logic languages have been refined in order to avoid many non-deterministic choices and the implementation of functional as well as logic languages has been improved considerably. This has demonstrated that functional logic languages can be efficiently implemented in a very similar way to (pure) functional or (pure) logic languages.

Curry is a functional logic programming language that combines the best ideas of existing declarative languages such as Haskell [HPJW92] and SML [MTH90] (functional languages), Gödel [HL94] and λProlog [NM88] (logic languages), and ALF [HS91], Babel [KA96] and TOY [CRLFSH97] (functional logic languages) in an elegant, practical, comprehensive, and seamless way. More specifically, Curry includes higher-order features, a type system, a module system, modern evaluation strategies, non-determinism, (encapsulated) search, partial data structures, existential variables, constraints, and declarative I/O.

UPV-Curry is a novel implementation of Curry which has been developed by the ELP group at the Technical University of Valencia (UPV) and is publicly available from the following URL:

`http://www.dsic.upv.es/users/elp/upv-curry/upv-curry.html`

This report is organized as follows. Section 2 summarizes the essentials of UPV-Curry. Section 3 introduces the UPV-Curry development system. Section 4 proposes some guidelines for working with the interpreter and different examples. Section 5 points out some differences between Curry and UPV-Curry. Section 6 summarizes some information sources for Curry and UPV-Curry. Appendix A and B show the BNF grammar of UPV-Curry and the standard prelude of built-in data types, operators and functions, respectively.

2 Technical summary of the main features

UPV-Curry provides an almost complete implementation of the Curry language, in accordance with its current definition [HAK$^+$98]. The main features are:

- Higher order functions (including lambda abstractions and Haskell sections).
- Curry constraint solver.
- Basic built-in types: Booleans, Integers, Characters, Strings, Lists, and Tuples.

- A polymorphic Hindley/Milner type system.
- A complete implementation of the monadic I/O declarative model.

The UPV-Curry interpreter is written in SICStus Prolog v3.6 and consists of about 250 Prolog clauses (2700 lines of code). This system is based on an interpreter which provides:

- a small self-contained Curry implementation, which constitutes a portable stand-alone SICStus Prolog application.
- a read-eval-point loop for displaying the solution for each expression which is entered as an input to the interpreter.
- naïve browsing facilities which are able to obtain the type and the definitional tree (a data structure needed to define the operational model of Curry) of each function.
- a debugging module, which allows the user to trace the evaluation of goals.

In this report, we assume that the reader is acquainted with the syntax and main features of Curry. Thus, we will not present an overview of the language. More details can be found in [HAK+98].

3 The UPV-Curry environment

This section provides the basic concepts about interacting with the UPV-Curry system.

UPV-Curry needs no previous installation. It runs under SICStus Prolog v3.x and is entered by simply typing:

```
?- load('upv-curry').
```

under the SICStus Prolog interpreter. The system can also be entered by typing in a shell command:

```
sicstus -l upv-curry
```

Once the user has started the interpreter, the following startup message screen shows up:

```
***********UPV-Curry interpreter*******************

Loading prelude...
Parsing...
Checking Types...
Generating definitional trees...
```

```
Commands:
   :load "<File>"       - Load Curry file in interpreter [:l]
   <Expression>         - Evaluate expression
   :debug <Expression> - Debug expression evaluation    [:d]
   :deftree <Function> - Show function deftree          [:e]
   :type <Expression>  - Show expression type           [:t]
   :cd "<Directory>"   - Change working directory       [:c]
   :pwd                - Show working directory         [:p]
   :help               - This help menu                 [:h]
   :quit               - Exit interpreter               [:q]

 :? refers to the short form of each command (only first character)

>
```

Interaction with the interpreter is done by typing in after the command prompt `>`. Here we can type either a menu command or an evaluable expression. We refer to Section 4 for a description of the commands listed above.

3.1 Expressions

Concerning ground expressions (without variables), UPV-Curry is just like a calculator. For example, if you type an arithmetic expression, the corresponding value (if any) is returned.

```
> 1+1
Solution: [ 2 ] 0 ms
> 1+2+3+4+5+6+7+8+9+10
Solution: [ 55 ] 40 ms
```

As with interpreters for functional languages, arbitrary (nonarithmetic) ground expressions can be evaluated.

```
> (not True) || False
Solution: [ False ] 0 ms
```

However, whenever non ground expressions are tried, the logic part of Curry becomes apparent since the corresponding substitution is returned.

```
> [1,x] ++ [3,4] =:= [1,2,y,4]
Solution: [ {x=2, y=3} ] 0 ms
```

3.2 Programs

The interpreter provides a set of built-in types, a set of operators and a set of functions that allow the user to type in basic expressions. Nevertheless, the user may also be generally interested in defining new types, functions and operators to solve certain problems. A program contains these new datatypes and functions that must be loaded into UPV-Curry in order to make them available to the user.

For example, suppose you have typed in the following program[1]:

```
data Person = Mary | John | Carla | Peter | Andrew

mother :: Person -> Person
mother eval flex
mother Peter = Carla
mother John  = Carla
mother Carla = Mary
```

which is stored into the file `mother.cur`. Here we have a definition of the new datatype `Person` and the definition of the function `mother`. After loading the program into UPV-Curry by typing the following command:

```
> :load "Examples/mother.cur"
```

you can evaluate the following expressions:

```
> mother Peter
Solution: [ Carla ] 0 ms
> mother John
Solution: [ Carla ] 0 ms
> mother Carla
Solution: [ Mary ] 0 ms
> mother Andrew
No solution. 0 ms
```

You can also ask for the evaluation of more complex expressions involving variables.

```
> mother x
Solution: [ {x=Peter} Carla ] 0 ms
Solution: [ {x=John} Carla ] 0 ms
Solution: [ {x=Carla} Mary ] 0 ms
```

Note that computed solutions in Curry have the general form: `[` {*substitution*} *value* `]`, which is simpler for the evaluation of ground expressions (`[` *value* `]`) and constraints (`[` {substitution} `]`), see Section 3.1.

4 Starting UPV-Curry

In UPV-Curry you are able to load programs, get help, retrieve information about the program components and exit the interpreter. These actions, and more, are done by using commands which start with a colon '`:`' followed by the name (or only the first letter) of the command itself. The commands you can type into the UPV-Curry system are the following:

[1] UPV-Curry does not provide any editing environment.

Exit UPV-Curry :quit

If you are in UPV-Curry, you type the command:

```
> :quit
```

to end the UPV-Curry session.

Help :help

This command shows the initial UPV-Curry help menu. If you type `:help` in the UPV-Curry prompt, the screen will show the following text.

```
Commands:
   :load "<File>"       - Load Curry file in interpreter [:l]
   <Expression>         - Evaluate expression
   :debug <Expression> - Debug expression evaluation    [:d]
   :deftree <Function> - Show function deftree          [:e]
   :type <Expression>  - Show expression type           [:t]
   :cd "<Directory>"   - Change working directory       [:c]
   :pwd                - Show working directory         [:p]
   :help               - This help menu                 [:h]
   :quit               - Exit interpreter               [:q]

 :? refers to the short form of each command (only first character)

>
```

Current working directory :pwd

This command presents the current working directory in UPV-Curry. The user's programs and text files are loaded using the working directory as the default path. The following is an example of the execution of this command:

```
> :pwd
/users/elp/upv-curry
>
```

Change working directory :cd "<Path>"

This command allows the user to change the working directory to a different one. If the user has to load, read or write a file stored in a directory other than the current one, then the command `cd` will allow the user to change the current directory. The following execution shows how the command `cd` changes the current working directory.

```
> :pwd
/users/elp/upv-curry
> :cd "Examples"
> :pwd
/users/elp/upv-curry/Examples
>
```

Loading a program :load "<File>"

This command loads the program file "File" which must contain a valid Curry program. Loading includes parsing the program text, checking type consistency and generating the definitional trees (one per function). It is necessary to write the file name between quotes. For example, to load the following Curry program stored in the file `lists.cur`, which is stored in the current working directory:

```
-- Concatenation of two lists
append :: [t] -> [t] -> [t]
append eval flex
append []     ys = ys
append (x:xs) ys = x:append xs ys

-- Naive reverse of all list elements
rev :: [t] -> [t]
rev []     = []
rev (x:xs) = append (rev xs) [x]

-- List membership using concatenation
member :: t -> [t] -> Bool
member e xs | append _ (e:_) =:= xs = True
```

you have to type in the following load command:

```
> :load "lists.cur"
```

and the system will show the following text indicating that the program file has been correctly loaded:

```
Parsing...
```

```
Checking Types...
Generating Definitional Trees...append rev member
```

If the file to be loaded is in another directory, the user can change the working directory with the command `cd`:

```
> :cd "Examples"
```

then the current working directory changes:

```
> :pwd
/users/elp/upv-curry/Examples
```

and finally you can enter the load command:

```
> :load "lists.cur"
Parsing...
Checking Types...
Generating Definitional Trees...append rev member
```

Another option is to specify the path in the load command:

```
> :load "Examples/lists.cur"
Parsing...
Checking Types...
Generating Definitional Trees...append rev member
```

Show definitional tree	`:deftree <function>`

This command shows the definitional tree [Ant92] for the function entered. This definitional tree is sent to the output with an indentation process that allows the user to quickly understand the branches of the definitional tree.

The following is an example of how to see the definitional tree of the function `append`:

```
> :deftree append
append:
branch((append x var2),1,flex) ->
 rule((append [] x) = x)
 rule((append (x:var2) var3) = (x:(append var2 var3)))
```

It is not necessary to take into account these definitional trees and its evaluation annotations in order to use UPV-Curry. They are automatically generated and managed by the system. We refer the reader to [HAK+98] to get more information about the role that definitional trees and evaluation annotations play in the operational semantics of Curry.

Show type **:type <expression>**

This command shows the type of the expression entered. The expression can be either a function name (to see the complete type of a function) or a function call with partial or complete data (showing the appropriate type).

The following example shows how the user can see the type of the function `append`:

```
> :type append
[var1] -> [var1] -> [var1]
```

The command `type` accepts any expression, and not just a function name. Some examples are:

```
> :type append [x]
[var1] -> [var1]
> :type append [1]
[Int] -> [Int]
> :type append [1] [2]
[Int]
```

Evaluate expression **<expression>**

Any expression entered will be evaluated. Before the evaluation, the expression is checked for syntactic correctness and type consistency.

The solutions are shown as they are found, showing the substitution (if any), the computed expression (if any) and the time spent (in ms) to compute each solution.

Here are some of the functional-like goals which are accepted and evaluated by the interpreter for the program `lists.cur`.

```
> append [1,2,3,4,5] [6,7,8,9,0]
Solution: [ [1,2,3,4,5,6,7,8,9,0] ] 0 ms
> rev [1,2,3,4,5,6,7,8,9,0]
Solution: [ [0,9,8,7,6,5,4,3,2,1] ] 140 ms
> member 1 [1,2,3,4,5,6,7,8,9,0]
Solution: [ True ] 40 ms
>
```

Due to the functional-logic semantics included in UPV-Curry, you can also evaluate expressions containing ocurrences of free variables:

```
> member x [1,2,3]
Solution: [ {x=1} True ] 70 ms
Solution: [ {x=2} True ] 30 ms
Solution: [ {x=3} True ] 40 ms
```

```
>
```

Or even evaluate constrained goals whose computed solutions not contain a value part:

```
> append x y =:= [1,2]
Solution: [ {x=[], y=[1,2]} ] 20 ms
Solution: [ {x=[1], y=[2]} ] 10 ms
Solution: [ {x=[1,2], y=[]} ] 10 ms
>
```

The following goals cannot be accepted by UPV-Curry due to syntactic or type errors.

```
> John mother Carla
Syntactic error, incorrect expression found
> 1+'a'
Type error, type Char is not correct when expected type Int
```

Note that an expression such as:

```
mother James
```

that can be interpreted by the user as an error (treating `James` as a constructor not specified in the `People` type), is really treated as a call to the function `mother` with the free variable `James`.

The user can press "`Ctrl+C`" at any time while the evaluation is in process in order to stop the evaluation. If pressed, SICStus Prolog will show the message `Prolog interruption`. By pressing the `a` key (which raises a SICStus Prolog abort command), UPV-Curry is set to manage the Prolog interruption, showing the following message.

```
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
```

This menu allows the user to either switch between normal and trace mode or to stop the evaluation (see the command `debug`).

Evaluate with debugging	:debug <expression>

The user can trace the evaluation process by using the debug mode. In order to do this, the command `debug` must be written before the expression to be evaluated:

```
:debug member x [1,2]
```

After each evaluation step, the system shows the following options:

```
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
```

and prompts the user to either stop the evaluation process (`a`), continue the evaluation without any pause (`c`), or continue with the next evaluation step (any other key). As remarked above, the user can press the "`Ctrl+C`" keys during the evaluation process, enabling the above control options. If the "`t - tracing in debug mode`" option is selected and the evaluation is not in debug mode, the evaluation process changes to debug mode as in the command `debug`.

The following sequences illustrate the debugging process for the evaluation of the expression `member x [1,2]` (the <enter> key is pressed in response to each '>' prompt):

```
> :debug member x [1,2]
-------------------------------------------------------
[ (member x [1,2]) ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append var9 (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append [] (x:var11)) =:= [1,2]) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
```

```
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((x:var11) =:= [1,2]) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((x =:= 1) & (var11 =:= [2])) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} ((success ) & (var11 =:= [2])) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} (var11 =:= [2]) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} ((var12 =:= 2) & (var13 =:= [])) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
```

```
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} ((success ) & (var13 =:= [])) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} (var13 =:= []) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} (success ) \=> True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=1} True ]
[ ((append (var12:var13) (x:var11)) =:= [1,2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
Solution: [ {x=1} True ] 80 ms
-------------------------------------------------------
[ ((var12:(append var13 (x:var11))) =:= [1,2]) \=> True ]
-------------------------------------------------------
```

```
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((var12 =:= 1) & ((append var13 (x:var11)) =:= [2])) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((success ) & ((append var13 (x:var11)) =:= [2])) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append var13 (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append [] (x:var11)) =:= [2]) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((x:var11) =:= [2]) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
```

```
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((x =:= 2) & (var11 =:= [])) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=2} ((success ) & (var11 =:= [])) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=2} (var11 =:= []) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=2} (success ) \=> True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ {x=2} True ]
[ ((append (var14:var15) (x:var11)) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
```

```
t - tracing in debug mode
(any other key) - next evaluation step
>
Solution: [ {x=2} True ] 70 ms
-------------------------------------------------------
[ ((var14:(append var15 (x:var11))) =:= [2]) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((var14 =:= 2) & ((append var15 (x:var11)) =:= [])) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((success ) & ((append var15 (x:var11)) =:= [])) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append var15 (x:var11)) =:= []) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append [] (x:var11)) =:= []) \=> True ]
[ ((append (var16:var17) (x:var11)) =:= []) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
```

```
-------------------------------------------------------
[ ((x:var11) =:= []) \=> True ]
[ ((append (var16:var17) (x:var11)) =:= []) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((append (var16:var17) (x:var11)) =:= []) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
-------------------------------------------------------
[ ((var16:(append var17 (x:var11))) =:= []) \=> True ]
-------------------------------------------------------
Curry Debug Control ---
a - abort
c - continue without debug
t - tracing in debug mode
(any other key) - next evaluation step
>
```

As the reader can see, the representation of conditional functions is translated to an internal representation using the function symbol `\=>`. The different states are shown using different text lines, owing to multiple instantiations.

5 Differences between Curry and UPV-Curry

With regard to the Curry model [HAK+98], some absences in the UPV implementation are:

- Modules.
- Encapsulated search.
- `choice` evaluation annotation.
- Local declarations (except let expressions allowing for the use of free variables).
- Interfaces to external functions and external constraint solvers.

6 Links to further information

For the latest description of the language Curry, see [HAK$^+$98]. More information about the development of the language can be found in the URL:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

For a complete description of UPV-Curry, see [EALV98]. UPV-Curry is free and available at the URL:

`http://www.dsic.upv.es/users/elp/upv-curry/upv-curry.html`.

Suggestions and comments are welcome and can be e-mailed to:

`sescobar@dsic.upv.es`.

A BNF Grammar

The grammar of UPV-Curry is a subset of the grammar described in appendix C of [HAK$^+$98]. The auxiliary symbols: *TypeConstrID*, *TypeVarID*, *DataConstrID*, *TypeConstrID*, *VariableID*, and *FunctionID* are the distinct types of identifiers; they are formed with an initial letter followed by letters, numbers or the underscore symbol "_". The auxiliary symbol *InfixOpID* is a string of the following symbols: "`~!#$%^&*+-=<>?./|`" or an identifier *id* enclosed within quotes: '*id*'.

Curry includes different syntactic modes, described in Section C.1 of [HAK$^+$98], to control the differentiation step between variables and the rest of the symbols. However, UPV-Curry only allows one possibility, the *free* mode (which is the default Curry mode) that does not constrain the case of identifiers.

A.1 Layout

The Curry language provides the use of layout information to define the structure of `where` and `let` blocks (see Section C.2 of [HAK$^+$98]). The indentation of a symbol is the column number indicating the start of this symbol. The indentation of a line is the indentation of its leftmost symbol. A Curry program is viewed as a declaration list of a `where` block inside a module declaration. Therefore, each program declaration is an element of this declaration list and has to be indented. However, although UPV-Curry does not provide for modules and local declarations, each program declaration has to be indented (in analogy to Curry). Hence, in the UPV-Curry system each program declaration is assumed to occur in column 1, whereas end-of-file token is in column 0.

In Curry, declaration lists in `where` and `let` blocks are enclosed within curly brackets ({ }) and the elements are separated by (`;`). Instead of defining these lists using curly brackets and semicolons, a Curry programmer should specify these lists by indentation. After the `where` or `let` keyword, the indentation of the first declaration is remembered, and subsequent declarations must have the same indentation as the first declaration. Lines with only whitespaces or an indentation greater than the indentation of the first declaration continue the declaration started in the preceeding line. Lines with an indentation less than the indentation of the first declaration terminates the entire list.

Program	::=	*Block*
Block	::=	{ [*PragmaDeclarationL*`;`]
		[*FixityDeclarationL*`;`]
		BlockDeclarationL }
PragmaDeclarationL	::=	*PragmaDeclaration* [`;` *PragmaDeclarationL*]
PragmaDeclaration	::=	`pragma` *PragmaOption*
PragmaOption	::=	`flex` | `rigid` | `optmatch`
FixityDeclarationL	::=	*FixityDeclaration* [`;` *FixityDeclarationL*]
FixityDeclaration	::=	*FixityKeyword Natural InfixOpIDL*
FixityKeyword	::=	`infixc` | `infixl` | `infixr`
Natural	::=	*Digit* [*Natural*]
Digit	::=	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
InfixOpIDL	::=	*InfixOpID* [`,` *InfixOpIDL*]
BlockDeclararionL	::=	*BlockDeclaration* [`;` *BlockDeclarationL*]
BlockDeclaration	::=	*DataDeclaration* | *FunctionDeclaration*
DataDeclaration	::=	`data` *TypeDeclaration*
TypeDeclaration	::=	*TypeConstrID* [*TypeVarIDL*] `=` *ConstrDeclarationL*
TypeVarIDL	::=	*TypeVarID* [*TypeVarIDL*]
ConstrDeclarationL	::=	*ConstrDeclaration* [`|` *ConstrDeclarationL*]
ConstrDeclaration	::=	*DataConstrID* [*TypeExprL*]
TypeExprL	::=	*TypeExpr* [*TypeExprL*]
TypeExpr	::=	*TypeConstrID* [*TypeExprL*]
	|	*TypeVarID*
	|	`()`
	|	`(`*TypeExprs*`)`
	|	`[`*TypeExpr*`]`
	|	*TypeExpr* `->` *TypeExpr*
	|	`Bool` | `Int` | `Char` | `String`
TypeExprs	::=	*TypeExpr* [`,` *TypeExprs*]
FunctionDeclaration	::=	[*Signature*] [*EvalAnnot*] *Equation*
Signature	::=	*FunctionName* `::` *TypeExpr*
FunctionName	::=	`(` *InfixOpID* `)`
	|	*FunctionID*
EvalAnnot	::=	*FunctionName* `eval` *Annotation*
Annotation	::=	`flex` | `rigid`
Equation	::=	*FunLHS Pattern* `=` *Expr*
	|	*FunLHS Pattern CondExprs*
FunLHS	::=	*FunctionName PatternL*
	|	*Pattern InfixOpID Pattern*
Pattern	::=	*VariableID*
	|	`_`

$$
\begin{array}{rcl}
 & \mid & \mathit{DataConstrID} \\
 & \mid & \texttt{()} \\
 & \mid & \texttt{(}\mathit{Patterns}\texttt{)} \\
 & \mid & \texttt{(}\ \mathit{DataConstrID}\ \mathit{PatternL}\ \texttt{)} \\
 & \mid & \texttt{(}\ \mathit{Pattern}\ \texttt{:}\ \mathit{Pattern}\ \texttt{)} \\
 & \mid & \texttt{[]} \\
 & \mid & \texttt{[}\mathit{Patterns}\texttt{]} \\
\mathit{PatternL} & ::= & \mathit{Pattern}\ [\mathit{PatternL}] \\
\mathit{Patterns} & ::= & \mathit{Pattern}\ [\texttt{,}\mathit{Patterns}] \\
\mathit{CondExprs} & ::= & \texttt{|}\ \mathit{Expr}\ \texttt{=}\ \mathit{Expr}\ [\mathit{CondExprs}] \\
\mathit{Expr} & ::= & \texttt{let}\ \mathit{VariableIDL}\ \texttt{free in}\ \mathit{Expr} \\
 & \mid & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr} \\
 & \mid & \backslash\ \mathit{PatternL}\ \texttt{->}\ \mathit{Expr} \\
 & \mid & \mathit{Expr}\ \mathit{InfixOpID}\ \mathit{Expr} \\
 & \mid & \mathit{FunctExpr} \\
\mathit{VariableIDL} & ::= & \mathit{VariableID}\ [\texttt{,}\ \mathit{VariableIDL}] \\
\mathit{FunctExpr} & ::= & [\mathit{FunctExpr}]\ \mathit{BasicExpr} \\
\mathit{Exprs} & ::= & \mathit{Expr}\ [\texttt{,}\mathit{Exprs}] \\
\mathit{BasicExpr} & ::= & \mathit{VariableID} \\
 & \mid & \texttt{()} \\
 & \mid & \texttt{(}\mathit{Expr}\texttt{)} \\
 & \mid & \texttt{(}\mathit{Expr}\texttt{,}\mathit{Exprs}\texttt{)} \\
 & \mid & \texttt{[]} \\
 & \mid & \texttt{[}\mathit{Exprs}\texttt{]} \\
 & \mid & \mathit{Bool} \mid \mathit{Int} \mid \mathit{Char} \mid \mathit{String}
\end{array}
$$

B Standard Prelude

This appendix shows the UPV-Curry prelude which is loaded during the startup. This prelude is a subset of the Curry prelude file showed in appendix B of [HAK+98]. All of the prelude function definitions of Curry that use encapsulated search have been dropped. Also, all of the prelude function definitions of Curry that use local declarations have been rewritten in a simpler way without using local declarations. These changes were done due to the absence of these features in UPV-Curry.

```
{-
--* Introduces information that cannot be included
in syntactic analysis, but can help the user.
-}

-- Infix operators declarations

infixl 9 ., !!
infixl 7 *, /, `div`, `mod`
```

```
infixl 6 +, -
infixr 5 ++
infixc 4 ==, <, >, =<, >=
infixl 3 &&
infixl 2 ||
infixr 1 >>, >>=, =:=
infixr 0 &, &>


-- Constraints

-- Equational constraint
(=:=) :: a -> a -> Constraint

-- Always solvable constraint
success :: Constraint

-- Concurrent conjunction of constraints
(&) :: Constraint -> Constraint -> Constraint

-- Sequential conjunction  of constraints
(&>) :: Constraint -> Constraint -> Constraint
c1 &> c2 | c1 = c2


-- Some standard combinators:

-- Function composition
(.)             :: (b -> c) -> (a -> b) -> (a -> c)
f . g           = \x -> f (g x)

-- Identity
id              :: a -> a
id x            = x

curry           :: ((a,b) -> c) -> a -> b -> c
curry f a b     = f (a,b)

uncurry         :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

flip            :: (a -> b -> c) -> b -> a -> c
flip f x y      = f y x


-- Boolean values

data Bool  = True | False

-- Sequential conjunction
```

```
(&&)       :: Bool -> Bool -> Bool
True  && x = x
False && _ = False

-- Sequential disjunction
(||)       :: Bool -> Bool -> Bool
True  || _ = True
False || x = x

-- Negation
not        :: Bool -> Bool
not True   = False
not False  = True

-- Otherwise function
otherwise  :: Bool
otherwise  = True


-- Pairs

--* data (a,b) = (a,b)

fst        :: (a,b) -> a
fst (x,_)  = x

snd        :: (a,b) -> b
snd (_,y)  = y


-- Unit type

--* data () = ()


-- Lists

--* data [a]          = [] | a : [a]

head              :: [a] -> a
head (x:_)        = x

tail              :: [a] -> [a]
tail (_:xs)       = xs

-- Concatenation
(++)              :: [a] -> [a] -> [a]
[]     ++ ys      = ys
(x:xs) ++ ys      = x : xs++ys
```

```
-- List length
length           :: [a] -> Int
length []        = 0
length (_:xs)    = 1 + length xs

-- List index (subscript) operator, head has index 0
(!!)             :: [a] -> Int -> a
(x:_)  !! 0      = x
(_:xs) !! n | n>0 = xs !! (n-1)

-- Map a function on a list
map              :: (a->b) -> [a] -> [b]
map _ []         = []
map f (x:xs)     = f x : map f xs

-- Accumulate all list elements
foldr            :: (a->b->b) -> b -> [a] -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

-- Filter elements in a list
filter           :: (a -> Bool) -> [a] -> [a]
filter _ []      = []
filter p (x:xs)  = if p x then x : filter p xs
                          else filter p xs

-- Join two lists to one list of pairs
zip              :: [a] -> [b] -> [(a,b)]
zip []     []    = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys

-- Concatenate a list of lists into one list
concat           :: [[a]] -> [a]
concat l         = foldr (++) [] l

-- Return prefix of length n
take             :: Int -> [a] -> [a]
take _ []    = []
take n (x:xs) = if n==0 then [] else x : take (n-1) xs

-- Return suffix without first n elements
drop             :: Int -> [a] -> [a]
drop _ []    = []
drop n (x:xs) = if n==0 then (x:xs) else drop (n-1) xs

-- Return longest prefix with elements satisfying a predicate
takeWhile         :: (a -> Bool) -> [a] -> [a]
takeWhile _ []    = []
takeWhile p (x:xs) = if p x then x : takeWhile p xs else []
```

```
-- Return suffix without takeWhile prefix
dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile _ []     = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs


-- Conversion functions between characters and their ASCII values
ord :: Char -> Int
chr :: Int -> Char


-- Convert a term into a printable representation

show :: a -> String


-- Types of primitive functions and predicates

(+)  :: Int -> Int -> Int
(-)  :: Int -> Int -> Int
(*)  :: Int -> Int -> Int
div  :: Int -> Int -> Int
mod  :: Int -> Int -> Int
(<)  :: Int -> Int -> Bool
(>)  :: Int -> Int -> Bool
(<=) :: Int -> Int -> Bool
(=>) :: Int -> Int -> Bool


-- Monadic IO

--* data IO a -- conceptually: World -> (a,World)

(>>)          :: IO a -> IO b        -> IO b
(>>=)         :: IO a -> (a -> IO b) -> IO b
putChar       :: Char -> IO ()
getChar       :: IO Char
done          :: IO ()
return        :: a -> IO a
readFile      :: String -> IO String
writeFile     :: String -> String -> IO ()

putStr        :: String -> IO ()
putStr []     = done
putStr (c:cs) = putChar c >> putStr cs

putStrLn      :: String -> IO ()
putStrLn cs   = putStr cs >> putChar '\n'

getLine       :: IO String
```

```
getLine        = getChar >>= \c ->
                   if c == '\n' then return []
                                else getLine >>= \cs -> return (c:cs)
```

References

[Ant92] S. Antoy. Definitional trees. In *editors Proc. of the 3rd International Conference on Algebraic and Logic Programming ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[CRLFSH97] R. Caballero Roldán, F.J. López Fraguas, and J. Sánchez-Hernández. User's Manual for Toy. Technical Report SIP-9757, UCM, 1997.

[EALV98] S. Escobar, M. Alpuente, S. Lucas, and G. Vidal. Un intérprete de Curry con tipos polimórficos y E/S declarativa monádica (in spanish). Technical Report DSIC-II/40/98, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, November 1998.

[HAK+98] M. Hanus, S. Antoy, H. Kuchen, F.J. López-Fraguas, and F. Steiner. Curry An Integrated Functional Logic Language (November 1998). Available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

[HL94] P. Hill and J.W. Lloyd. *The Gödel Programming Language*. The MIT Press, Cambridge, MA, 1994.

[HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *Sigplan Notices*, 27(5), 1992.

[HS91] M. Hanus and A. Schwab. The Implementation of the Functional-Logic Language ALF. FB Informatik, Univ. Dortmund, 1991.

[KA96] H. Kuchen and J. Anastasiadis. Higher Order Babel: Language and implementation. In *editors Proc. 5th International Workshop on Extensions of Logic Programming ELP'96*, pages 193–207. Springer LNAI 1050, 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of ML*. The MIT Press, 1990.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of Fifth Int'l Conf. on Logic Programming ICLP'88*, pages 810–827. The MIT Press, Cambridge, MA, 1988.