

The Quick Start Guide

Roman Lacko

May 19, 2014

1 Introduction

This document serves as a **quick start** guide for demo applications attached to the *Mercury* library. More detailed description of the libraries is in the **Documentation** folder.

2 MercuryParser

This application serves as a demonstration of usage of the library. It incorporates **Interpreter** with return type **dynamic**, so it can handle all basic types like **string** and **integer** at the cost of speed.

To run the program, first open the **CommandPrompt** from the directory with the application (in Microsoft Windows, **Shift** key and **Right Mouse Click**, then select **Open command window here**).

```
MercuryParser /g <name> [other arguments]
```

If run with no arguments, the application will proceed to the **Interactive** mode (explained later). In both cases, the Mercury **BasicScanner** is used before parsing.

2.1 Command-line mode

Available command-line arguments are as follows:

```
/h /help /?
```

shows help for both command-line arguments and interactive commands.

```
/interactive
```

enforces interactive mode (ignores all other arguments)

```
/g /grammar <name>
```

loads a grammar from a file specified as the **name** option argument; this argument **must** be specified

```
/a /actions <name>
```

loads rewrite rules from a file specified as the **name** option argument; this argument is optional

`/c /chart`
 will display the chart of the parser after parsing
`/t /trees`
 will display parse trees
`/i /interpret`
 will interpret parse trees according to rules provided by the **actions** argument;
 if rules are not specified, an error will be reported

In addition, at least one of **chart**, **trees** and **interpret** **must** be specified. These arguments are also available in their UN*X-like format, that is, one-character arguments prefixed with `-` and long arguments prefixed with `-`. For instance, `-c` is equal to `/c` and `-interactive` is equal to `/interactive`.

When application is running, no prompt is be shown. The program will read input characters until it encounters the **End-Of-File** character, which can be inserted into the Windows console as keyboard shortcut **Ctrl-Z** and **Enter** in the new line. The whole input will be considered as one since new line characters will be ignored.

Example run:

```
E:\Attachments\Executables>MercuryParser /g ../Examples/catalan.grm
                                     /a ../Examples/catalan.rwr /t /i
```

```
x x x x
^Z
0: [S [S [S x] [S x]] [S [S x] [S x]]]
1: [S [S x] [S [S x] [S [S x] [S x]]]]
2: [S [S x] [S [S [S x] [S x]] [S x]]]
3: [S [S [S x] [S [S x] [S x]]] [S x]]
4: [S [S [S [S x] [S x]] [S x]] [S x]]

0: ((x x) (x x))
1: (x (x (x x)))
2: (x ((x x) x))
3: ((x (x x)) x)
4: (((x x) x) x)
```

```
E:\Attachments\Executables>
```

2.2 Interactive mode

Running without command-line arguments or with `/interactive` argument will force the application to run in interactive mode. The application shows the prompt `>` as it awaits a command.

A string before prompt specifies which components are used for analysis.

> no grammar nor rewrite rules are loaded
Parse> only grammar is loaded, interpretation is not available,
Interpret> both grammar and rewrite rules are loaded

All commands in the interactive mode are prefixed with the `:` character. Other inputs will be attempted to be analysed. To insert the colon as input, use escaped sequence `:. The following commands are available in the interactive mode:`

```
:a :actions
    shows loaded rewrite rules
:c :chart
    shows chart of the last parsing
:empty runs analysis on an empty string ""
:g :grammar
    shows loaded grammar
:? :h :help
    shows available commands with a brief description
:i :interprets
    shows interpretations created by the last analysis
:l :load <grammar> (<rules>)
    loads a grammar from a file specified as grammar; the second argument specifies
    rewrite rules and is optional
:log ix (<file>)
    writes the interpreter's log of the ix-th interpretation in the console window or
    (if specified) into the file specified as file
:q :quit exits the application
:r :result
    shows brief result of the last analysis
:t :trees
    shows trees created by the last parsing
:v :version
    prints versions of libraries and the application
```

If any command is specified with incorrect arguments, it will fail and show an error message.

Example run of the MercuryParser:

```
E:\Attachments\Executables>MercuryParser /interactive
Mercury Tools :: Bottom-Up Chart Parser
=====

Use :h (:help) to invoke help.
> :l ../examples/rpn.grm ../examples/rpn.rwr
Could not find file 'E:\Attachments\examples.rpn.rwr'.

> :l ../examples/rpn.grm ../examples/rpn.rwr
Files loaded successfully

Interpret> ((1 + 2) * (4 - 5))
Result:          Success
Parser time:      00:00.00
Interpreter time: 00:00.25
Total time:       00:00.26
Edges in chart:   58
Derivation trees: 1
Interpretations:  1

Interpret> :t
0: [Expr '(' [Expr '(' [Expr [@Number 1]] [@Op '+']
    [Expr [@Number 2]] ')'] [@Op '*'] [Expr '('
    [Expr [@Number 4]] [@Op '-'] [Expr [@Number 5]] ')']
    ')']

Interpret> :i
0: * + 1 2 - 4 5

Interpret> :q

E:\Attachments\Executables>
```

3 PerfWrapper

The **Performance Wrapper** application can be used to analyse performance of the library. With no arguments specified, it will run and start analysis of built-in resource files. The actual resource is specified in the source code as this mode is designed for *Microsoft Profiling Tools* and it is more convenient to change options directly than in command line.

The following command-line options will run performance tests on specific grammars and rewrite rules.

`/a <min> <max> <logs>`
runs the **Test A** on a grammar and files specified in the `Examples\catalan.grm` and `Examples\catalan.rwr`. Optional arguments `min` and `max` specify the minimum and maximum number of tokens. It is not recommended to specify the `max` value more than 14. The `logs` flag (values `true` or `false`) indicates whether interpreter logs should be created in the `Logs` directory. Default values for these arguments are `8 15 false`.

`/b`
runs run **Test B** using the **MQNLParse**r application on inputs specified in the `Examples\bc.in`.

Examples:

```
E:\Attachments\Executables>PerfWrapper /a 6 10
```

```
Mercury Tools :: Performance Wrapper
```

```
=====
```

```
Running performance test A
```

```
Loading grammar ...
```

```
Loading actions ...
```

```
Running dummy analysis to enforce JIT... (done)
```

```
Running tests ... (might take a while)
```

```
(100 % | cl 10) Interpreting (4862 trees)...
```

```
=====
```

ix	tc	nc	ttp	tti	ltp	lti
6	42	17	0 ms	1 ms	0 ns	23 ns
7	132	20	0 ms	2 ms	0 ns	15 ns
8	429	23	2 ms	13 ms	4 ns	30 ns
9	1430	26	8 ms	9 ms	5 ns	6 ns
10	4862	29	28 ms	67 ms	5 ns	13 ns

```
=====
```

```
E:\Attachments\Executables>
```

Meaning of the columns of the table is as follows: `ix` is the number of tokens, `tc` is the number of trees, `nc` is the number of nodes per tree, `ttp` and `1tp` is parsing time of all trees and one tree respectively, similarly for `tti` and `1ti` for interpretation.

4 MQNLParse

The MQNLParse application demonstrates the use of the Mercury library for translating English phrases into MotiveQuery expressions.

The application is interactive and features only the following commands:

```
grammar  shows the grammar
rwrules  shows the rewrite rules
exit     exits the application
```

All other inputs will be attempted to be analysed. The very first query is usually slower than others due to **JIT** and lazy evaluation featured in the F# programming language.

Example:

```
E:\Attachments\Executables>MQNLParse.exe
Mercury Tools :: Motive Query Analyzer
=====

Grammar rules:      287
Interpreter rules: 111
> Any residue that contains at least seven zinc atoms.

  1 result (T 277 ms) (P 29 ms) (I 97 ms)
~ Filter[ResidueSet[],Lambda[(m0),LessEqual[7,
  Count[AtomSet[Zn],Symbol["m0"]]]]]

> exit

E:\Attachments\Executables>
```

4.1 Language extension

This section is also available in the **Appendix A.2** of the thesis.

To extend the language that the application recognizes one could use operators defined in the `DeclOp` module and add new rules to the `LanguageData` module. These operators allow the grammar to be brief and compiled with the entire application without the need of additional parsing. The `Language` module also contains *metafunctions* that can create several rules at once using phrasal patterns (see below). There are many of these operators, but the code is documented and function of each operator is sufficiently explained.

Examples:

Phrases (concept example)

```
pp2 NOUN          PREP    ADJ    OBJ
pp2 "AminoAcid"   "PAgent" "Charge" "NCharge"
```

creates a preposition phrase with object of the form DET NOUN PREP ENUM(ADJ) DET OBJ. It can match sentences like "(any) (amino acid) (with) (polar or aromatic) (charge)". Parenthesis separate the constituents, i.e. PREP = with.

Meta functions

```
metaEnum Q                == EnumQ -> Q ; EnumQ -> Q Conj EnumQ
metaEnum @AtomName
```

constructs enumeration for Q where Conj represents conjunctions. The example with @AtomName can for instance match zinc or oxygen.

```
lex NAME      [X; Y; ...] == NAME -> X | Y | ...
lex "NParent" [ "parent"; "structure"; "motive"; "motif" ]
```

is a convenience structure. The name suggests that it should be used to construct the *lexicon*.

Grammar operators

```
"A" ---> ["B"; "C"]          == A -> B C
"A" -->> [["B"; "C"]; ["D"]] == A -> B C | D
```

Element operators

```
!~ "A"                == A          %% constant
!~~ [ "A"; "B" ]      == A|B        %% constant
!&"x"                 == #x::NT     %% variable
"x" &~ "A"            == #x:A        %% variable + alt.
"x" &~~ [ "A"; "B" ]   == #x:A|B     %% variable + alt.
!~"x" |-- [ !~"A"; !~"B" ] == [x A B] %% structure
w(); eps(); any()     == *; e; NTe  %% wildcards, epsilon
```

Action operators

```
f |< [x; y; z]      == (f x y z)  %% action call
f ^|< x             == (f x)      %% action call
!< f               == (f)        %% action call
!* "x"             == #x         %% variable parameter
ip 3 ; rp 3.0       == 3 ; 3.0   %% integer and real const.
bp true ; sp "xyz"  == true ; xyz %% bool and string const.
ev' "x"            == (eval #x)  %% shorthand
```

Complex examples

```
!"Filter" |-- [!&"m"; w(); "p" &~ "Predicate"]
          ==> (filter |< [ev' "m"; ev' "p"])
[Filter #m * #p:Predicate] ==> (filter (eval #m) (eval #p))

!~"@Number" |-- [!&"n"]
          ==> (toqbv ^|< parsei ^|< name ^|< !*"n")
[@Number #n] ==> (toqbv (parseinteger (name #x)))

!"Count" |-- [!~"NCount"; "ms" &~ "EnumMSeq"; !~"NCount"]
          ==> (cntlmb ^|< qbeor ^|< ev' "ms")
[Count NCount #ms:EnumMSeq NCount] ==> (cntlmb (qbeor (eval #ms)))
```

More details about the syntax are provided in the comments of the application's source code.