λlama. - Lambda Calculus Interpreter

Giannakopoulos Anastasios
`sdi0900053@di.uoa.gr`

Aftsidis Sergios
`sdi0900052@di.uoa.gr`

Deparment of Informatics and Telecommunications
University of Athens
Greece

February 25, 2015

Contents

1 Introduction

This document describes the λlama λ-calculus interpreter. This is a project implemented by Giannakopoulos Anastasios and Aftsidis Sergios as an assignment in an undergraduate course, "*Principles of Programming Languages*". This document will guide you through everything you will need to use the interpreter, from the really basic aspects (compiling and running the program, running simple (or even more advanced) λ-terms) to theoretical / technical specifics about the implementation (evaluation strategy / implementation details).

1.1 Description

λlama is a λ-calculus interpreter which (though minimal) is quite powerful. It has a simple interface which makes it easy and quick to learn.

1.2 Supported Features

1.2.1 Basic Features

λlama supports all the basic λ-calculus features. It follows the commonly used λ-term syntax, which is

```
term ::== var                     // variable
       |  (term) (term)           // application
       |  λ(var).(term)           // abstraction
```

Here, *var* represents a variable, which is a sequence of letters starting with a lower-case letter (e.g. x, xYY, x_01 but **not** X). The cases of *variable* and *application* are really straightforward, so lets focus on *abstraction*. A legal abstraction syntax can be

$$\backslash(var).(BODY)$$

where "var" is the abstraction variable, eg. x and $BODY$ is the body of the λ-term. So, a valid λ-term would be $\backslash x.x$. You can of course compose more complex terms, such as $\backslash x.x\backslash y\ .y\ z$

Q: How is this term interpreted?
Following the standard λ-calculus conventions, this is interpreted as $\backslash x.(\ x\ (\backslash y.(y\ z)))$.
(The abstraction rule extends the furthest possible to the right)

Q: How can a user choose the how the term will be displayed?
By entering the command : *par on* the user specifies that the term will be displayed using all the possible parenthesis. By choosing : $par\ off$, only the required parenthesis will be displayed.

1.2.2 Numerals

λ-calculus supports numbers, known as *Church Numerals*. λlama supports *Church Numerals*, but also integers in the form we know them. For example, one can represent the number 3 both as $(\backslash f.\backslash x\ f(f(f\ x)))$ and 3.

1.2.3 Identifiers

λlama also supports user-defined identifiers. Identifiers can be used to make λ-calculus more practical, as they provide a level of abstraction from λ-terms to a more humanly readable context. So, the interpreter also processes expressions like

$$And\ True\ False$$

which outputs $False$ (*Actually*, the output is *0*, but the Church Numeral for 0 is $\backslash f.\backslash x.\ x$, which is also the *False* identifier).

One can really see the power of identifiers in examples like the following:

$$Add\ (Minus\ 34\ 5)\ (Mult\ 3\ 12)$$

Such a λ-term would take up more than two lines to write, and it is really easy to make a mistake while composing it.

Q: How can a user define custom identifiers?
By using the *store* command. After entering : *store* (followed by "*Enter*") the user can assign custom Identifiers (starting with an uppercase letter), following the syntax *Identifier = Term.*

Q: Where is the list with the pre-defined identifiers?
The full list with the identifiers pre-loaded with λlama is in the file *./src/.llama_aliases*

1.2.4 Operators

λlama also supports numeric operators, with their infix notation. So the above expression can be also written as $(34 - 5) + (3 * 12)$. Also, the operators follow the conventional operator priority, so the above expression can also be written as $34 - 5 + 3 * 12$. In order to account for operators, the grammar has been modified as such

```
term ::== var                        // variable
          ...
          |   (term) OP (term)       // OP = operator
```

Supported Operators:

- **Arithmetic Operators**: "+" , "-" , "*" , "/" ,"∧" (exponential), "<" ,"≤", ">", "≥
- **Logical Operators**: "&&" (And), "||" (Or), "==" (Equal), "!=" (Not Equal)

2 Installation

2.1 Dependencies

λlama requires *Flex* and *Bison* to compile entirely from source. In case these utilities are not installed, you can build directly from the files generated from them (see section below). Also *g++* is required to compile the source files. This implementation has been developed to for Linux systems.

2.2 Compilation

To use the application, you must first extract and compile it. To extract, navigate to the directory where the compressed file is and execute

```
# tar -xf llama.tar.gz
```

For easier compilation, a *makefile* is included. To compile using the automatically created *Flex* and *Bison* files *(recommended)*, use the commands

```
# cd ./llama
# make llama
```

If, however, you want to create again the automatically generated files from *Flex* and *Bison*, you can use the commands

```
# cd ./llama/src
# make parser.cpp
# make parser.hpp
# make tokens.cpp
# cd ..
# make llama
```

In order to make the interpreter more functional, we have included a wrapping utility, which adds some really convenient functionality, e.g. using the up and down arrow keys you can see previously entered commands, using the left - right arrow keys you can navigate through the term you are composing, entering a right a parenthesis will highlight the matching left one etc. This utility must also be extracted and compiled. To do so, enter the following commands

```
# tar -xf rlwrap-0.30.tar.gz
# make rlwrap
```

3 Usage

This section describes the usage of λlama . Wherever the *executable* file is referenced, you should replace it with the name of the executable you are using. If you are using the wrapped version, then replace it with λlama *−rlwrap*, otherwise with λlama

If at any time the user needs information, there is a help message brought up by the : *help* command.

3.1 Command-line input

Executing the command

```
# ./executable
```

will start the interpreter in command line mode. In this mode you can type the terms you want and interpret them real time.

3.2 Input through file

Executing the command

```
# ./executable inputFile
```

will start the interpreter in file input mode. In this mode the terms from *inputFile* will be interpreted first, and then you are redirected to command line mode.

3.3 Tracing

λlama provides a tracing functionality. This allows the user to trace all the reductions done while evaluating a λ-term one by one. To enable tracing, use the command : *trace on*. To disable it, use : *trace off*. When tracing, the interpreter each time executes a reduction and then waits for user input (*Enter*).

4 Implementation Details

4.1 Parsing - Semantics - Representation

As previously mentioned, lexing and parsing are handled using the *Flex* and *Bison* lexer and parser. *Flex* recognizes the input and returns the corresponding tokens to *Bison*, which in turn creates the syntax tree. This tree represents the λ-term we are evaluating and this is the data structure on which we are performing evaluations.

4.2 Evaluation Strategy

There are several evaluation strategies in λ-calculus , but only *Normal Order Evaluation* strategy (see here) guarantees that we will definitely reach the term's normal form (*if* it exists). So, this strategy is implemented. This strategy can be roughly described as: ***Always*** *choose the leftmost β-redex* first.