The Journal of Open Source Education

ChAmElEoN: A Customizable Language and Suite of Interpreters in Python for Teaching Concepts of Programming Languages

Saverio Perugini[1] **and Jack L. Watkin**[1]

1 Department of Computer Science, University of Dayton, 300 College Park, Dayton, Ohio 45469-2160 USA

DOI: 10.21105/jose.00056

Software

- Review
- Repository
- Archive

Submitted: 20 May 2019
Published: 10 August 2019

Summary

ChAmElEoN is a programming language, in the tradition of EOPL (Friedman, Wand, & Haynes, 2001), for teaching students the concepts and implementation of computer languages through the development of a series of interpreters for it written in Python. We describe its syntax and semantics, the educational aspects involved in the implementation of a variety of interpreters for it, its malleability, and student feedback to inspire its use for teaching languages (Perugini & Watkin, 2018).

Statement of Need

While there are multiple implementation languages used to develop interpreters for typical courses in programming languages using the interpreter-based approach (e.g., Racket (Krishnamurthi, 2012,Krishnamurthi (2008)), LISP (Queinnec, 2003), or Java (Fossum, 2014)), **ChAmElEoN is, to the best of our knowledge, the only EOPL-style interpreter (Friedman et al., 2001) for use in programming languages courses developed in Python**—a language whose use is going rapidly in computer science courses at all levels.** While the PLCC (Programming Language Compiler-Compiler) system (Fossum, 2014) is written in Python, it generates parsers, interpreters, and compilers in Java. What sets the interpreter-based approach in ChAmElEoN apart from the others, and in particular (Friedman et al., 2001), is the use of Python—an approachable, practical, and widely-used programming language—as the implementation language. Thus, this software fills a much-needed void in the software available to the programming languages education community.

Introduction

The ChAmElEoN programming language, inspired by (Friedman et al., 2001), is a language for teaching students the concepts and implementation of computer languages through the development of language interpreters in Python. In particular, in the course of their study of programming languages, students implement a variety of an environment-passing interpreters for ChAmElEoN, in the tradition of (Friedman et al., 2001), in Python.

There are multiple benefits from incrementally implementing language interpreters. First, students are confronted with one of the most fundamental truths of computing: the interpreter for a computer language is just another program" (Friedman et al., 2001). Second, once a language interpreter is established as just another program, students realize quickly that implementing a new concept, construct, or feature in a computer language amounts to little more than a few lines of code in the interpreter. Third, students learn the causal relationship between a language and its interpreter. In other words, they realize that an interpreter for a language explicitly defines the semantics of the language it interprets. The consequences of this realization are compelling: students are mystified by the drastic changes they can affect in the semantics of implemented language by changing only a few lines of code in the interpreter—sometimes as little as one line (e.g., using dynamic scoping rather than static scoping, or using lazy evaluation as opposed to eager evaluation).

```
<program> ::= <expression>
<program> ::= <statement>
<expression> ::= <number> | <string>
<expression> ::= <identifier>
<expression> ::= if <expression> <expression> else <expression>
<expression> ::= let {<identifier> = <expression>}{+} in <expression>
<expression> ::= <primitive> ({}^{+(,)})
<primitive> ::= + | - | * | inc1 | dec1 | zero? | eqv? | read | array |
arrayreference | arrayassign
<expression> ::= <function>
<expression> ::= let* {<identifier> = <expression>}+ in <expression>
<function> ::= fun ({identifier}{*(,)}) <expression>
<expression> ::= (<expression> {<expression>}{*(,)})
<expression> ::= letrec {<identifier> = <function>}+ in <expression>
<expression> ::= assign! <identifier> = <expression>
<statement> ::= <identifier> = <expression>
<statement> ::= writeln (<expression>)
<statement> ::= {{<statement>}{+(;)}}
<statement> ::= if <expression> <statement> else <statement>
<statement> ::= while <expression> do <statement>
<statement> ::= variable {<identifier>}{+(,)} ; <statement>
```

Figure 1: The grammar in EBNF for the ChAmElEoN programming language (from (Perugini & Watkin, 2018)).

Grammar

The grammar in EBNF for ChAmElEoN (version 3.3.a) is given in Figure 1. ChAmElEoN can be used as a functional, expression-oriented language (Savage, 2018) or as a statement-oriented language or both. To use it as an expression-oriented language, use the *<program>* ::= *<expression>* grammar rule; to use it as an imperative, statement-oriented language, use the *<program>* ::= *<statement>* rule. User-defined functions are first-class entities in ChAmElEoN. This means that a function can be the return value of an expression (i.e., an expressed value), bound to an identifier and, thus, stored in the environment of the interpreter (i.e., a denoted value), and passed as an argument to a function. Notice from the production rules in Figure 1, ChAmElEoN supports side effect (through variable assignment) and arrays. The primitives `array`, `arrayreference`, and `arrayassign` create an array, dereference an array, and update an array, respectively. While we have multiple versions of ChAmElEoN, each supporting varying concepts, in version 3.3.a

Expressed Value	=	Integer U String U Closure
Denoted Value	=	Reference to an Expressed Value

Thus, akin to Java or Scheme, all denoted values are references, but are implicitly dereferenced. For more details of the language, we refer the reader to Perugini & Watkin (2018)].

Table 1: Configuration options in ChAmElEoN (from (Perugini & Watkin, 2018)).

Type of Environment	Representation of Environment	Representation of Functions	Scoping Methods	Environment Binding	Parameter-passing Mechanism
Named	Abstract Syntax	Abstract Syntax	Static	Deep	By-value
Nameless[1]	List of Vectors	-expression	Dynamic	Shallow	By-reference
	-expression			Ad-hoc	By-value-result
					By-name (lazy eval.)
					By-need (lazy eval.)

Installation

To install the environment necessary for running ChAmElEoN, follow these steps:

1. Install Python v3.4.6 or later.
2. Install PLY v3.9 or later.
3. Clone the latest ChAmElEoN repository.

Example Ubuntu 18.04 installation with `apt` package manager:

```
$ sudo apt install python3
$ sudo apt install python3-pip
$ sudo python3 -m pip install ply
$ git clone https://bitbucket.org/chameleoninterpreter/chameleon-interpreter-in-py
```

[1]Not all implementation options are available for use with the nameless environment. * * * * *

Repository Structure and Setup

The release versions of the ChAmElEoN interpreters in Python are available in a Git repository in BitBucket at https://bitbucket.org/ChAmElEoNinterpreter/ChAmElEoN-interpreter-in-python-release/src/master/ (Perugini & Watkin, n.d.).

The repository is organized into the following main sub-directories, described in detail below, indicating the recommended order in which instructors and, thus, students should explore them:

`0.0_FRONTEND` (the front end syntactical analyzer for the language) `1.0_INTRODUCTION` (interpreters with support for local binding and conditionals) `2.0_INTERMEDIATE` (interpreters with support for functions and closures) `3.0_ADVANCED` (interpreters with support for a variety of parameter-passing mechanisms, including lazy evaluation)

```
Each sub-directory contains a ```README.md``` file indicating the recommended orde

# How to Use ChAmElEoN in a Programming Languages Course

## Module 0: Front End  (Scanner and Parser)

The first place to start is with the front end of the interpreter which contains t
a scanner/parser generator for Python-and have been tested in Python 3.4.6. For th

| Description
| ---------------------------------------------------------------------------------
| ChAmElEoN Installation instructions
| Scanner for ChAmElEoN
| Parser for ChAmElEoN

## Module I: Introduction (Local Binding and Conditionals)

Given the parser, students start by implementing only primitive operations, save f

Adding a support for a new concept or feature to the language typically involves a

Students, thus, start by adding support for conditional evaluation and local bindi
choose.

In what follows, each directory corresponds to the different (progressive) version

| Interpreter Description
| ---------------------------------------------------------------------------------
| Simple interpreter with primitives
| Interpreter with local binding and conditionals


## Configuring the Language

Table 2 enumerates the configuration options available in ChAmElEoN for aspects of

The configuration file (in ```chameleonconfig.py```) allows the user to switch bet

------------------
```

$ pwd chameleon-interpreter-in-python-release $ cd pass-by-value-recursive $ cat chameleonconfig.py … … closure_closure = 0 #static scoping our closure representation of closures asr_closure = 1 #static scoping our asr representation of closures python_closure = 2 #dynamic scoping python representation of closures **closure_switch** = asr_closure # for lexical scoping #**closure_switch** = python_closure # for dynamic scoping

closure = 1 asr = 2 lovr = 3 **env_switch** = lovr

detailed_debug = 1 # full stack trace through python exception simple_debug = 2 # chameleon interpreter output only **debug_mode** = simple_debug $

```
------------------


* * * * *
Table 2: Design choices and implemented concepts in progressive versions of ChAmEl

| **Version of ChAmElEoN**     | **1.0**| **2.0**    | **2.1**              | **3.
| ---------------------------- | :----: | :--------: | :------------------: | :---
| **Expressed Values**         | ints   | ints U cls | ints U cls           | ints
| **Denoted Values**           | ints   | ints U cls | refs. to expr'd vals. | refs
| **Rep. of Env.**             | N/A    | 3 possible | 3 possible           | 3 po
| **Rep. of Functions**        | N/A    | 2 possible | 2 possible           | 2 po
| **Rep. of References**       | N/A    | N/A        | abstract syntax rep. | abst
| **Local Binding**            | ↑let↑  | ↑let↑      | ↑let↑                | ↑let
| **Conditionals**             | ↓cond↓ | ↓cond↓     | ↓cond↓               | ↓con
| **Non-recursive Functions**  |  ×     | ↑fun↑      | ↑fun↑                | ↑fun
| **Recursive Functions**      |  ×     | ↑fun↑      | ↑fun↑                | ↑fun
| **Scoping**                  |  ×      | lexical    | lexical              | lexi
| **Env. Bound to Closure**    | N/A    | deep       | deep                 | deep
| **References**               | N/A    | N/A        | :ballot_box_with_check: | :b
| **Parameter Passing**        | N/A    | ↑by value↑ | ↑by reference↑       | ↑by
| **Side Effects**             |  ×     |    ×       | ↑assign↑             | ↓mul
| **Statement Blocks**         | N/A    |   N/A      | N/A                  | :bal
| **Repetition**               | N/A    |   N/A      | N/A                  | ↓whi

* * * * *

At this point, students can also explore implementing dynamic scoping as an altern

## Module II: Intermediate (Non-recursive Functions and Closures)

Next, students implement recursive functions, which require a modified environment
a purely functional language-and explored the use of multiple configuration option

| Interpreter Description
| -------------------------------------------------------------------------------
| Interpreter with non-recursive functions using pass-by-value
| Nameless Environment Interpreter with non-recursive functions using pass-by-valu
| Interpreter with recursive functions using pass-by-value
| Interpreter with recursive functions using pass-by-value

## Module III: Advanced (Parameter Passing, including Lazy Evaluation)

Next, students start slowly to morph ChAmElEoN, through its interpreter, into an i
```

```
| Interpreter Description
| ------------------------------------------------------------------------------
| Interpreter for ChAmElEoN supporting variable assignment (i.e., side effect)
| Interpreter for ChAmElEoN supporting arrays
| Interpreter for ChAmElEoN supporting pass-by-value-result parameter passing
| Interpreter for ChAmElEoN supporting pass-by-reference parameter passing
| Interpreter for lazy ChAmElEoN supporting pass-by-(name/need) parameter passing
| Interpreter for lazy ChAmElEoN version 3.2 with lazy lets
| Interpreter for lazy ChAmElEoN version 3.2.a with lazy primitives and if primiti
| Interpreter for ChAmElEoN with more imperative language features than side effec

------------------

# Test Coverage

We analyzed the code coverage of these tests cases using a tool for analyzing
code test coverage called ```Coverage.py```
(https://coverage.readthedocs.io/en/v4.5.x/).  This tool allows you to run our
test cases above and then output a report that shows the percentage of the
lines of code missed by those test cases.  For instance, the following table is
the output of running this tool on our test cases for the ChAmElEoN interpreter
(version 3.0.b) supporting arrays (i.e., ```./arrays/tests.cham```).

| Source Code Filename           | Statements Missed | Total Statements | Coverage
| ------------------------------ | -----------------:| ----------------:| ---------
| ```chameleonclosure.py```      | 0                 |   27             | 100%
| ```chameleonconfig.py```       | 0                 |   11             | 100%
| ```chameleonenv.py```          | 1                 |   41             |  98%
| ```chameleoninterpreter.py```  | 15                |  200             |  92%
| ```chameleonlex.py```          | 6                 |   43             |  86%
| ```chameleonlib.py```          | 5                 |   24             |  79%
| ```chameleonparse.py```        | 29                |  122             |  76%
| ```chameleonreferences.py```   | 6                 |   43             |  86%
| ```parsetab.py```              | 0                 |   18             | 100%
| **TOTAL**                      | 62                |  533             | **88%**

Many of the missed lines here are exceptions that are unreachable because there
are not any syntax errors in the test cases and because there is code blocked
off for different modes of the interpreter.

# Example Usage: Non-interactively and Interactively (CLI)

Once students have some experience implementing language interpreters, they can be
unsupported in the interpreter.  For instance, prior to supporting recursive funct

------------------
```

$ pwd chameleon-interpreter-in-python-release $ $ cd pass-by-value-non-recursive $ $ # running the interpreter non-interactively $ $ cat recursionUnbound.cham let sum = fun (x) if zero?(x) 0 else +(x, (sum dec1(x))) in (sum 5) $ $ ./run recursionUnbound.cham Line 2: Unbound identifier 'sum' $ $ cat recursionBound.cham let sum = fun (s, x) if zero?(x) 0 else +(x, (s s,dec1(x))) in (sum sum, 5) $ $ ./run recursionBound.cham 15 $ $ # running the interpreter interactively (CLI) $ $ ./run ChAmElEoN> let sum = fun (x) if zero?(x) 0 else +(x, (sum dec1(x))) in (sum 5)

Runtime Error: Line 2: Unbound Identifier 'sum'

ChAmElEoN> let sum = fun (s, x) if zero?(x) 0 else +(x, (s s,dec1(x))) in (sum sum, 5)

15 $ "' ——————

Other example programs, including an example more faithful to the tenants of object-orientation, especially encapsulation, are available in our Git repository. These programs demonstrate that we can create object-oriented abstractions from within the ChAmElEoN language.

Conclusion

What sets the interpreter-based approach in ChAmElEoN apart from the others, and in particular (Friedman et al., 2001), is the use of Python—an approachable, practical, and widely-used programming language—as the implementation language. The use of ChAmElEoN is integrated into a programming languages textbook—titled *Programming Languages: Concepts and Implementation*—which is available free and by request on a trial basis for educators interested in adopting this approach (contact lead author at saverio@udayton.edu for more information). A sample course outline of topics, including course notes, through the textbook is available online at http://academic.udayton.edu/SaverioPerugini/pl.

Acknowledgments

This material is based on the following paper: Perugini, S. & Watkin, J.L. (2018). ChAmElEoN: A customizable language for teaching programming languages. *Journal of Computing Sciences in Colleges*, **34**(1), 44—51. USA: Consortium for Computing Sciences in Colleges. ACM Digital Library. (Available at https://dl.acm.org/citation.cfm?id=3280498.)

This material is based upon work supported by the National Science Foundation under Grant Numbers 1712406 and 1712404. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Fossum, T. (2014). PLCC: A programming language compiler compiler. In *Proceedings of the 45th ACM technical symposium on computer science education (SIGCSE)* (pp. 561–566). New York, NY: ACM Press. doi:10.1145/2538862.2538922

Friedman, D., Wand, M., & Haynes, C. (2001). *Essentials of programming languages* (Second.). Cambridge, MA: MIT Press.

Krishnamurthi, S. (2008). Teaching programming languages in a post-Linnaean age. *ACM SIGPLAN Notices*, *43*(11), 81–83. doi:10.1145/1480828.1480846

Krishnamurthi, S. (2012). *Programming languages: Application and interpretation.*

Perugini, S., & Watkin, J. (2018). ChAmElEoN: A customizable language for teaching programming languages. *Journal of Computing Sciences in Colleges*, *34*(1), 44–51.

Perugini, S., & Watkin, J. (n.d.).

Queinnec, Q. (2003). *LISP in small pieces.* Cambridge, UK: Cambridge University Press.

Savage, N. (2018). Using functions for easier programming. *Communications of the ACM*, *61*(5), 29–30. doi:10.1145/3193776