

University of Dayton eCommons

Computer Science Faculty Publications

Department of Computer Science

10-2018

ChAmElEoN: A Customizable Language for Teaching Programming Languages


Saverio Perugini

University of Dayton, sperugini1@udayton.edu

Jack L. Watkin

University of Dayton

Follow this and additional works at: https://ecommons.udayton.edu/cps_fac_pub

 Part of the [Graphics and Human Computer Interfaces Commons](#), and the [Other Computer Sciences Commons](#)

eCommons Citation

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 Article is brought to you for free and open access by the Department of Computer Science at eCommons. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of eCommons. For more information, please contact frice1@udayton.edu, mschlangen1@udayton.edu.

CHAMELEON: A CUSTOMIZABLE LANGUAGE FOR TEACHING PROGRAMMING LANGUAGES

Saverio Perugini and Jack L. Watkin
Department of Computer Science
University of Dayton
300 College Park
Dayton, Ohio 45469-2160
(937) 229-4079
saverio@udayton.edu

ABSTRACT

CHAMELEON is a programming language for teaching students the concepts and implementation of computer languages. 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.

INTRODUCTION

The CHAMELEON programming language, inspired by [3], is a language for teaching students the concepts and implementation of computer languages. In particular, in the course of their study of programming languages, students have implemented a variety of an environment-passing interpreters for CHAMELEON, in the tradition of [3], initially in Racket (Scheme) and, more recently, in Python.

The scanner and parser for CHAMELEON were developed using Python Lex-Yacc (PLY v3.9)—a scanner/parser generator for Python—and have been tested in Python 3.4.6. For the details of PLY, see <http://www.dabeaz.com/ply/>. The front end of our CHAMELEON interpreter in Racket is built using SLLGEN—a scanner/parser generator for Scheme.

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

Figure 1: The grammar in EBNF for the CHAMELEON programming language.

The grammar in EBNF for CHAMELEON (v4) is given in Figure 1. CHAMELEON can be used as a functional, expression-oriented language [7] or as a statement-oriented language or both. To use it as an expression-oriented language, use the $\langle program \rangle ::= \langle expression \rangle$ grammar rule; to use it as an imperative, statement-oriented language, use the $\langle program \rangle ::= \langle statement \rangle$ 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 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 4

Expressed Value = Integer \cup String \cup Closure
Denoted Value = Reference to an Expressed Value.

Thus, akin to Java or Scheme, all denoted values are references, but are implicitly dereferenced.

LEARNING LANGUAGES THROUGH INTERPRETERS

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” [3]. 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 seman-

tics 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).

Students start by implementing only primitive operations (see Figure 1; save for array manipulations). Then, students develop an `evaluate-expression` function which accepts an expression and an environment as arguments and evaluates the passed expression in the passed environment and returns the result. This function, which is at the heart of any interpreter, constitutes a large conditional structure based on the type of expression passed (e.g., a variable reference or function definition). Then students add support for conditional evaluation and local binding. Support for local binding requires a lookup environment which leads to the possibility of testing a variety of representations for that environment, as long as it adheres to the well-defined interface used by `evaluate-expression`. From there, students add support for non-recursive functions, which raises the issue of how to represent a function of which there are a host of options from which to choose. At this point, students can also explore implementing dynamic scoping as an alternative to the default static scoping. This amounts to little more than storing the calling environment, rather than the lexically enclosing environment, in the representation of the function. Next, students implement recursive functions, which require a modified environment. At this point, students have implemented CHAMELEON v2—a purely functional language—and explored the use of multiple configuration options for both aspects of the design of the interpreter as well as the semantics of implemented concepts (see Table 1).

Next, students start slowly to morph CHAMELEON, through its interpreter, into an imperative language by adding provision for side effect (e.g., through variable assignment). Variable assignment

Table 1: Configuration options in CHAMELEON.

Interpreter Design Options			Language Semantic Options		
Type of Environment	Representation of Environment	Representation of Functions	Scoping Method	Environment Binding	Parameter Passing Mechanism
Named	Abstract Syntax	Abstract Syntax	Static	Deep	By-value
Nameless ¹	List of Vectors λ -expression	λ -expression	Dynamic	Shallow Ad-hoc	By-reference By-value-result By-name (lazy eval.) By-need (lazy eval.)

requires a modification to the representation of the environment. Now, the environment must store references to expressed values, rather than the expressed values themselves. This raises the issue of implicit versus explicit dereferencing, and naturally leads to exploring a variety of parameter-passing mechanisms (e.g., pass-by-reference or pass-by-name/lazy evaluation). Finally, students close the loop on the imperative approach by eliminating the need to use recursion for repetition by instrumenting the language, through its interpreter, to be a statement-oriented, rather than expression-oriented, language. This involves adding support for statement blocks, while loops, and I/O operations.

The use of a scanner/parser generator facilitates this incremental development approach which leads to a malleable interpreter/language. Adding a new feature typically involves adding a new grammar rule and/or primitive, adding a new field to the abstract syntax representation of an expression, and adding a new case to the `evaluate-expression` function. This is theme of [3].

Configuring the Language

Table 1 enumerates the configuration options available in CHAMELEON for aspects of the design of the interpreter (e.g., choice of representation of referencing environment), as well as for the semantics of implemented concepts (e.g., choice of parameter-passing mechanism). As we vary the latter, we get a different version of the language (see Table 2).

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

Table 2: Design choices and implemented concepts in progressive versions of CHAMELEON. The symbol ↓ indicates that the concept is supported through its implementation in the defining language. The symbol ↑ indicates that the concept is implemented from first principles.

	Version of CHAMELEON	1	2	3	4
Design Choices	Expressed Values	ints	ints ∪ cls	ints ∪ cls	ints ∪ cls
	Denoted Values	ints	ints ∪ cls	refs. to expr'd vals.	refs. to expr'd vals.
	Rep. of Env.	N/A	3 possible	3 possible	3 possible
	Rep. of Functions	N/A	2 possible	2 possible	2 possible
	Rep. of References	N/A	N/A	ASR	ASR
Language Semantic Options	Local Binding	↑ let ↑	↑ let ↑	↑ let ↑	↑ let ↑
	Conditionals	↓ cond ↓	↓ cond ↓	↓ cond ↓	↓ cond ↓
	Non-recursive Functions	×	↑ fun ↑	↑ fun ↑	↑ fun ↑
	Recursive Functions	×	↑ fun ↑	↑ fun ↑	↑ fun ↑
	Scoping	N/A	lexical	lexical	lexical
	Env. Bound to Closure	N/A	deep	deep	deep
	References	×	×	√	√
	Parameter Passing	N/A	↑ by value ↑	↑ by reference ↑	↑ by value ↑
	Side Effects	×	×	↑ assign! ↑	↓ multiple ↓
	Statement Blocks	N/A	N/A	N/A	√
	Repetition	N/A	N/A	N/A	↓ while ↓

Once students have some experience implementing language interpreters, they can begin to discern how to use the language itself to support features currently unsupported in the interpreter. For instance, prior to supporting recursive functions in CHAMELEON, students can simulate support for recursion by passing a function to itself:

```

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)

```

Example CHAMELEON Program: A Simple Stack Object

Through an extension of the prior idea, even though CHAMELEON does not have support for object-oriented programming, students can use CHAMELEON to build object-oriented abstractions. For instance, the following CHAMELEON program, simplified for purposes for exposition, simulates the implementation of a simple stack class with two constructors (`new_stack` and `push`) and three observers/messages (`emptystack?`, `top`, `pop`). The output of this program is 3. The stack object is represented as a CHAMELEON closure.

```
let
  — constructor
  new_stack = fun ()
    fun(msg)
      if eqv?(msg, 1)
        -1 — error: cannot top an empty stack
      else
        if eqv?(msg, 2)
          -2 — error: cannot pop an empty stack
        else
          1 — represents true: stack is empty

  — constructor
  push = fun (elem, stack)
    fun (msg)
      if eqv?(msg,1) elem
      else if eqv?(msg,2) stack
      else 0

  — observers
  emptystack? = fun (stack) (stack 0)
  top = fun (stack) (stack 1)
  pop = fun (stack) (stack 2)
in
  let
    simplestack = (new_stack)
  in
    (top (push 3, (push 2, (push 1, simplestack))))
```

Other example programs, including an example more faithful to the tenants of object-orientation, especially encapsulation, are available in our Git repositories (see Table 3). These programs demonstrate that we can create object-oriented abstractions from within the CHAMELEON language.

Table 3: Links to versions of CHAMELEON interpreters in Python and Racket.

Language	BitBucket Link to Git Repository
Python	https://bitbucket.org/chameleoninterpreter/chameleon-interpreter-in-python-release/src/master/
Racket	https://bitbucket.org/chameleoninterpreter/chameleon-interpreter-in-racket-release/src/master/

STUDENT FEEDBACK

Students have found CHAMELEON interpreter-building helpful and fun, and to have educational merit.

Building the interpreter was helpful.

Implementing these concepts first hand is what makes this class so worthwhile.

I really liked taking a look at the interpreter, which is at the heart of programming languages. In fact, the interpreter is what defines the programming language.

I feel implementing concepts in a language is the best way to learn some of these tough concepts.

I would not ditch the interpreter, it is what ties many of the course themes together and it is where some of the more abstract concepts were concretely demonstrated.

My favorite module was definitely module three where we got to see how an interpreter comes together.

CONCLUSION

The interpreter-based approach toward learning programming languages is neither unique nor a panacea. Pedagogically, the interpreter and language survey approaches are essentially complements of each other in advantages and disadvantages. For a discussion of the differences and trade-offs, we refer the reader to [4]. A myriad of other approaches for teaching programming languages

have been tried and tested [1, 2, 5, 6, 8]. What sets the interpreter-based approach in CHAMELEON apart from the others, and in particular [3], 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. A sample course outline of topics, including course notes, through the textbook is available online at <http://academic.udayton.edu/SaverioPerugini/pl>. See Table 3 for links to our release versions of CHAMELEON interpreters in both Python and Racket.

ACKNOWLEDGMENTS

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. We thank Norman Bashias in the Department of Computer Science at the University of Dayton for providing comments on a draft of this paper.

REFERENCES

- [1] Adams, E., Baldwin, D., Bishop, J., English, J., Lawhead, P., and Stevenson, D. Approaches to teaching the programming languages course: A potpourri. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 299–300, New York, NY, 2006. ACM Press.
- [2] Fossum, T. PLCC: A programming language compiler compiler. In *Proceedings of the 45th ACM Technical Symposium on Com-*

puter Science Education (SIGCSE), pages 561–566, New York, NY, 2014. ACM Press.

- [3] Friedman, D., Wand, M., and Haynes, C. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, Second edition, 2001.
- [4] Krishnamurthi, S. Teaching programming languages in a post-Linnaean age. *ACM SIGPLAN Notices*, 43(11):81–83, 2008.
- [5] Lee, K. A framework for teaching programming languages. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 162–167, New York, NY, 2015. ACM Press.
- [6] Pombrio, J., Krishnamurthi, S., and Fisler, K. Teaching programming languages by experimental and adversarial thinking. In Lerner, B., Bodík, R., and Krishnamurthi, S., editors, *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, pages 13:1–13:9, 2017.
- [7] Savage, N. Using functions for easier programming. *Communications of the ACM*, 61(5):29–30, 2018.
- [8] Sheldon, M. and Turbak, F. An aspect-oriented approach to the undergraduate programming language curriculum. *ACM SIGPLAN Notices*, 43(11):124–129, 2008.