The Calysto Scheme Project

JAMES B. MARSHALL*, Sarah Lawrence College, USA
DOUGLAS S. BLANK*, Comet ML, Inc., USA

The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here. The abstract goes here.

CCS Concepts: • **Software and its engineering → General programming languages**; **Functional languages**; **Integrated and visual development environments**.

Additional Key Words and Phrases: Scheme, Python, Jupyter, whatever

ACM Reference Format:
James B. Marshall and Douglas S. Blank. 2023. The Calysto Scheme Project. 1, 1 (July 2023), 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

This paper describes the development of Calysto Scheme, a fully-featured implementation of the Scheme programming language written in Scheme, which can be automatically translated into other languages, such as Python and C# [13]. Although our path through the development of this system was circuitous, the completed project ended up being far more interesting (and useful) than we could have imagined.

The initial motivation for Calysto Scheme grew out of an earlier open-source project called Calico that was designed to be a multi-programming-language framework and learning environment for computing education [3]. The basic premise of this earlier project was to create a common architecture, user interface, and set of libraries for a variety of programming languages (see Figure 1).

Many engaging "pedagogical contexts" for learning about computer science have been developed, including media computation [6], gaming, AI and robotics [15], visualization, music, and art. However, these contexts often depend on a set of libraries developed for a specific programming language, which may constrain the choice of language if an instructor wishes to have students explore a particular learning context. Having a common framework separates the details of a specific language from other pedagogical goals.

One of the interesting aspects of Calico was that instead of having students learn a different IDE for each language under study, such as IDLE for Python, or DrRacket for Scheme, students could remain in the same IDE, but simply switch the programming language. This is similar in spirit to the way that one can switch languages in DrRacket. However, in Calico, supported languages need not share much at all with each other.

At the time (2007), Microsoft had embarked on a somewhat-related goal. They were actively developing what they called the Dynamic Language Runtime (DLR) as part of the .NET framework

*Both authors contributed equally to this work.

Authors' addresses: James B. Marshall, Sarah Lawrence College, 1 Mead Way, Bronxville, New York, 10708, USA, jmarshall@sarahlawrence.edu; Douglas S. Blank, Comet ML, Inc., New York, USA, doug.blank@gmail.com.

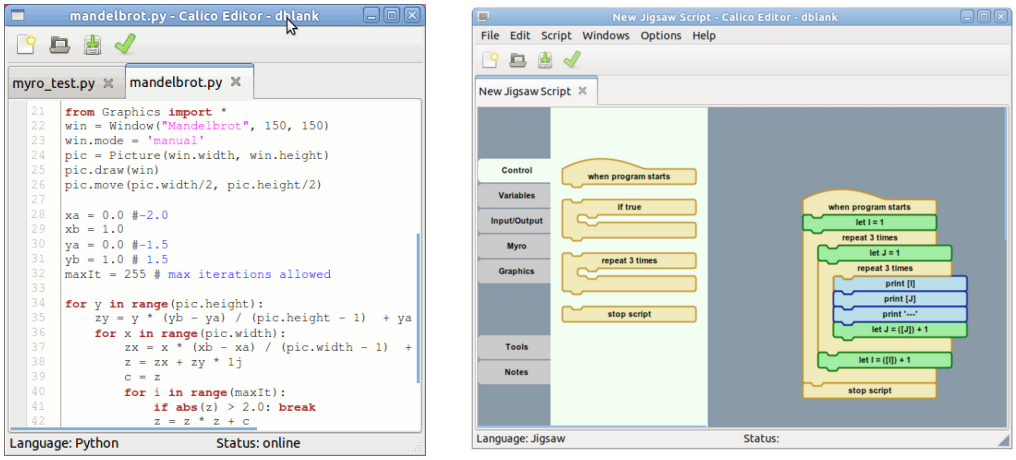

Fig. 1. The Calico interface, running Python (left), and Jigsaw, a visual block-based language (right).

[DLR]. The DLR abstraction layer created a common API and allowed one to create languages, such as IronPython and IronRuby [14], without having to rewrite the common parts for each language. Our earlier project adopted the DLR as its foundation, and used these Iron languages. In addition, the Calico team developed other languages to augment these, including a visual block-based language.

However, there was a gap in the languages available: there was no Scheme implementation that could work in this environment. Although the development of both IronLisp and IronScheme [12] had been attempted by other groups, they both ultimately failed for various reasons, including lack of support for tail-call optimization (TCO) in the DLR.[1]

Of course, the Scheme language can be implemented in languages without TCO. So, we set out to fill this void by developing a Scheme without using the DLR for internal function calls, but still allowing integration with the DLR for interoperation with other functionality (such as calling libraries). Thus, we began the development of what would become Calysto Scheme.

Once the core Scheme language was written in Scheme (as described below), the final conversion step to C# was relatively straightforward. At this point, we were able to add Scheme to our list of supported languages, including Python, F#, and Ruby. The code snippets in Figure 2 show demo scripts written in each language, all of which call functions from the same underlying graphics library. Each script creates a graphics window titled “Hello”, and draws a line between (0,0) and (100, 100). Although the calls to library functions in each language differ merely in their syntax, there are many deeper differences between the languages in terms of semantics.

Although we initially targeted C# as the implementation language, we eventually decided to replace C# with Python (see below). The core Calysto Scheme architecture remained the same, only the final transformation step needed to be changed to target Python rather than C#. The transformation to Python also included changing calls to C#’s DLR to calls to standard Python functions. The following section outlines the overall Calysto Scheme design pipeline.

2 CALYSTO SCHEME DESIGN

Calysto Scheme ensures full support for tail-call optimization, with no limit placed on the depth of the call stack. Unfortunately, languages such as C# and Python impose a maximum depth on

[1]IronScheme eventually did include at least some support for first-class continuations. However, it was incrementally added beginning in late 2008 [11] after we had begun building Calysto Scheme.

```
# Python Graphics Example
import Graphics
win = Graphics.Window("Hello")
line = Line((0,0), (100,100))
line.draw(win)

;; Scheme Graphics Example
(using "Graphics")
(define win (Graphics.Window "Hello"))
(define line (Graphics.Line (Graphics.Point 0 0) (Graphics.Point 100 100)))
(line.draw win)

// F# Graphics Example
module MyModule
let win = Graphics.Window("Hello")
let line = new Graphics.Line(new Graphics.Point(0,0), new Graphics.Point(100,100))
line.draw(win)

# Ruby Graphics Example
win = Graphics::WindowClass.new("Hello")
line = Graphics::Line.new(Graphics::Point.new(0,0), Graphics::Point.new(100,100))
line.draw(win)
```

Fig. 2. Calling the same graphics library functions from four different languages in Calico.

their recursion stack and do not support tail-call optimization. Therefore, our approach was to implement the Calysto Scheme interpreter in Scheme, and then automatically convert it, via a series of correctness-preserving program transformations, into low-level register machine code that does not rely on the recursion stack, which can then be directly transformed into Python (or C#) as the final step. The high-level Scheme version of the interpreter is written in continuation-passing style (CPS), with continuations initially represented as anonymous lambda functions. The continuations are then converted to a data structure representation (as lists), and from there passing information to functions via arguments is replaced by passing information via a set of global registers, which removes the reliance on the call stack. At this stage, the computation is driven by a single "trampoline" loop, essentially equivalent to a while loop [4, 5].

As an example of the transformation process, consider a simple recursive function that adds up the first n positive integers. We start with a version of this function written in CPS, using functional continuations, as shown below. For example, calling the top-level function `(sum 100)` returns 5050.

```
(define sum-cps
  (lambda (n k)
    (if (= n 0)
        (k 0)
        (sum-cps (- n 1)
          (lambda (value)
            (k (+ n value)))))))

;; top-level function
(define sum
  (lambda (n)
    (sum-cps n (lambda (value) value))))
```

```
;; global registers
(define n_reg 'undefined)
(define k_reg 'undefined)
(define value_reg 'undefined)
(define fields_reg 'undefined)
(define pc 'undefined)
(define final_reg 'undefined)

(define trampoline
  (lambda ()
    (if pc
        (begin
          (pc)
          (trampoline))
        final_reg)))

(define make-cont
  (lambda args
    (cons 'continuation args)))

(define apply-cont
  (lambda ()
    (let ((label (cadr k_reg))
          (fields (cddr k_reg)))
      (set! fields_reg fields)
      (set! pc label))))

(define <cont-1>
  (lambda ()
    (set! final_reg value_reg)
    (set! pc #f)))

(define <cont-2>
  (lambda ()
    (let ((n (car fields_reg))
          (k (cadr fields_reg)))
      (set! k_reg k)
      (set! value_reg (+ n value_reg))
      (set! pc apply-cont))))

(define sum-cps
  (lambda ()
    (if (= n_reg 0)
      (begin
        (set! value_reg 0)
        (set! pc apply-cont))
      (begin
       (set! k_reg (make-cont <cont-2> n_reg k_reg))
        (set! n_reg (- n_reg 1))
        (set! pc sum-cps)))))

;; top-level function
(define sum
  (lambda (n)
    (set! k_reg (make-cont <cont-1>))
    (set! n_reg n)
    (set! pc sum-cps)
    (trampoline)))
```

Fig. 3. Scheme register machine code

```
# global registers
n_reg = None
k_reg = None
value_reg = None
fields_reg = None
pc = None
final_reg = None

def trampoline():
    while pc:
        pc()
    return final_reg

def car(lst):
    return lst[0]

def cdr(lst):
    return lst[1:]

def cadr(lst):
    return car(cdr(lst))

def cddr(lst):
    return cdr(cdr(lst))

def make_cont(*args):
    return ("continuation",) + args

def apply_cont():
    global fields_reg, pc
    label = cadr(k_reg)
    fields = cddr(k_reg)
    fields_reg = fields
    pc = label

def cont_1():
    global final_reg, pc
    final_reg = value_reg
    pc = False

def cont_2():
    global k_reg, value_reg, pc
    n = car(fields_reg)
    k = cadr(fields_reg)
    k_reg = k
    value_reg = n + value_reg
    pc = apply_cont

def sum_cps():
    global value_reg, pc, k_reg, n_reg
    if n_reg == 0:
        value_reg = 0
        pc = apply_cont
    else:
       k_reg = make_cont(cont_2, n_reg, k_reg)
        n_reg = n_reg - 1
        pc = sum_cps

# top-level function
def sum(n):
    global k_reg, n_reg, pc
    k_reg = make_cont(cont_1)
    n_reg = n
    pc = sum_cps
    return trampoline()
```

Fig. 4. Python register machine code

We wrote a program that takes any Scheme program written in CPS, such as the above, and transforms the code into an equivalent register machine, with continuations represented as lists. The resulting register machine for `sum` is shown in Figure 3. Calling `(sum 100)` still returns 5050, as before.

At this stage, all functions other than the trampoline simply execute if-statements or update registers via assignment statements, without ever calling another function directly (except for low-level built-in primitives like `car` or `+`). This avoids building up chains of function calls. This code can then be directly converted into Python (see Figure 4). Since the Python version of `sum` is no longer constrained by the depth of the recursion stack, it can be called with arbitrarily large values of *n*.

In a similar fashion, we transform our Calysto Scheme interpreter from a high-level recursive CPS program written in Scheme into an equivalent low-level register machine written in Python, which does not grow Python's call stack.

There is an interesting aspect to creating a language in this manner: the implementation can be tested at each stage of the transformation process to ensure correctness. That is, the same suite of Scheme test programs can be run independently with the CPS, Data Structure, Register Machine, and Python implementations. In essence, the initial CPS definition serves as both a high-level language specification and an executable implementation of the language, from which the other three implementations are subsequently derived. This allows us to test each stage separately to catch bugs in the CPS specification and the transformation process itself.

The current version of Calysto Scheme requires an existing Scheme implementation to carry out the transformations from CPS to Data Structures, and from Data Structures to Register Machine. The final conversion from Scheme to Python is written in Python. We used Petite Chez Scheme in early development, and switched to Chez Scheme when it was made open source. In principle, it would be possible to make Calysto Scheme self-hosting (*i.e.*, Calysto Scheme could carry out the transformations itself). However, our transformation program currently relies on Chez Scheme's `syntax-rules` macro definition facility, which differs somewhat from the version of `define-syntax` implemented in Calysto Scheme.

3 SYNTACTIC EXTENSION

Calysto Scheme supports syntactic extension through its own version of `define-syntax`, which can define simple macros using standard list notation in conjunction with unification pattern matching variables that begin with the `?` character. For example, consider the following macro expansion rules for `and` and `or` expressions:

(and *exp*) → *exp*
(and exp_1 exp_2 exp_3 ...) → (if exp_1 (and exp_2 exp_3 ...) #f)

(or *exp*) → *exp*
(or exp_1 exp_2 exp_3 ...) → (if exp_1 #t (or exp_2 exp_3 ...))

Although `and` and `or` are already available in Calysto Scheme, in principle they could be implemented with the following recursive macro definitions:

```
(define-syntax and
  [(and ?exp) ?exp]
  [(and ?first-exp . ?other-exps) (if ?first-exp (and . ?other-exps) #f)])
```

```
(define-syntax or
  [(or ?exp) ?exp]
  [(or ?first-exp . ?other-exps) (if ?first-exp #t (or . ?other-exps))])
```

4 NONDETERMINISTIC BACKTRACKING

In their classic text *Structure and Interpretation of Computer Programs* (SICP) [1], Abelson and Sussman introduce the nondeterministic "amb" operator for automatic backtracking. We have incorporated this operator into Calysto Scheme as the special form (choose arg_1 arg_2 ... arg_n), which nondeterministically chooses one of its arguments to evaluate and returns the resulting value. From there, the computation proceeds normally, unless (choose) is subsequently invoked with no arguments, at which point the computation "fails" and immediately jumps back to the previous choose expression, whereby a different argument is chosen to evaluate next, and the computation restarts from that point with the new value. Many constraint-satisfaction problems can be elegantly solved using choose.

As an example illustrating both choose and define-syntax, suppose we wish to write a program to determine how to color a map of (a portion of) Western Europe using four distinct colors. We first define a function to nondeterministically return one of four possible colors:

```
(define choose-color
  (lambda ()
    (choose 'red 'yellow 'blue 'white)))
```

The color-europe program begins by nondeterministically assigning a color to each country on the map:

```
(define color-europe
  (lambda ()
    (let ([portugal (choose-color)]
          [spain (choose-color)]
          [france (choose-color)]
          [belgium (choose-color)]
          [germany (choose-color)]
          [luxembourg (choose-color)]
          [italy (choose-color)]
          [switzerland (choose-color)])
        ...)))
```

Next, we must apply the following constraint to each country: its chosen color must be different from that of all of its adjacent neighbors. For example, since Luxembourg is bordered by France, Belgium, and Germany, we could express its color constraint as follows:

```
(require (not (member luxembourg (list france belgium germany))))
```

The require function is a Calysto Scheme primitive similar to an assertion statement, which takes a boolean value as input and invokes (choose) if the input is false in order to force the program to backtrack to the most recent choice point, instead of raising an exception. However, a more elegant approach might be to define a new syntactic form called color that expresses this constraint in a very readable way:

```
(color luxembourg different from france belgium germany)
```

The Calysto Scheme macro definition for color, along with the complete program to determine a consistent map-coloring, is given below:

```
(define-syntax color
  [(color ?country different from . ?neighbors)
   (require (not (member ?country (list . ?neighbors))))])
```

```
(define color-europe
  (lambda ()
    (let ([portugal (choose-color)]
          [spain (choose-color)]
          [france (choose-color)]
          [belgium (choose-color)]
          [germany (choose-color)]
          [luxembourg (choose-color)]
          [italy (choose-color)]
          [switzerland (choose-color)])
      ;; apply the constraints
      (color portugal different from spain)
      (color spain different from france portugal)
      (color france different from spain italy switzerland belgium germany luxembourg)
      (color belgium different from france luxembourg germany)
      (color germany different from france switzerland belgium luxembourg)
      (color luxembourg different from france belgium germany)
      (color italy different from france switzerland)
      (color switzerland different from france italy germany)
      ;; return a coloring that satisfies the constraints
      (list (list 'portugal portugal)
            (list 'spain spain)
            (list 'france france)
            (list 'belgium belgium)
            (list 'germany germany)
            (list 'luxembourg luxembourg)
            (list 'italy italy)
            (list 'switzerland switzerland)))))
```

Calling `(color-europe)` returns the solution `((portugal red) (spain yellow) (france red) (belgium yellow) (germany blue) (luxembourg white) (italy yellow) (switzerland white))`. However, this is not the only valid solution; many other color combinations will work. We can force the program to backtrack to find another combination that satisfies the constraints, simply by calling `(choose)` as many times as we like, until all valid choices have been returned:

```
==> (choose)
((portugal red) (spain yellow) (france red) (belgium yellow) (germany blue)
 (luxembourg white) (italy blue) (switzerland yellow))
==> (choose)
((portugal red) (spain yellow) (france red) (belgium yellow) (germany blue)
 (luxembourg white) (italy blue) (switzerland white))
```

and so on. Eventually, after all possible combinations of choices that satisfy the constraints have been returned, any subsequent calls to `(choose)` will return the string “no more choices”, until a new `choose` expression is executed.

5 PYTHON / SCHEME INTEROPERATION

There are a number of ways that Calysto Scheme and Python can interoperate. For example, Python code can be directly evaluated from within Scheme. Scheme functions can be defined in an environment shared with Python, and then called by Python programs. And Python functions and libraries can be imported directly into Scheme and called from within Scheme programs.

Whereas in Scheme everything is an expression, Python makes a distinction between evaluation and execution. In Calysto Scheme, the function `python-eval` can be used to evaluate strings representing Python expressions, and `python-exec` can be used to execute Python statements.

```
(python-eval "1 + 2")
→ 3

(python-exec
"
def mypyfunc(a, b):
    return a * b
")

(python-eval "mypyfunc(2, 3)")
→ 6
```

The special form `func` turns a Scheme procedure into a Python function, and `define!` puts it into the shared environment with Python:

```
(define! mypyfunc2 (func (lambda (n) (* n n))))
(python-eval "mypyfunc2(3)")
→ 9
```

As a simple illustration of Calysto Scheme's ability to import and use functions from Python and its libraries, consider the following example, in which the elements of a nested Scheme list are summed by converting it into a Numpy array and then applying the Numpy `sum` function:

```
(import "numpy")

(let ([matrix3d '(((10 20 30) (40 50 60))
                  ((70 80 90) (100 110 120))
                  ((130 140 150) (160 170 180))
                  ((190 200 210) (220 230 240)))])
  (numpy.sum (numpy.array matrix3d)))
→ 3000
```

Python dictionaries can also be created and manipulated from within Calysto Scheme using a special syntax, as shown in the example interaction below:

```
==> (define d (dict '((apple : red) (banana : yellow) (lime : green))))
==> d
{'apple': red, 'banana': yellow, 'lime': green}
==> (get-item d 'apple)
red
==> (set-item! d 'apple 77)
==> d
{'apple': 77, 'banana': yellow, 'lime': green}
```

A more complex example that shows the power of combining these features is shown in Figure 5. This Scheme code sets up a series of machine learning experiments using the Numpy and Tensorflow libraries imported from Python, along with a third-party library called `comet_ml`

```
(import-as "tensorflow" "tf")
(import-as "numpy" "np")
(import "comet_ml")

;; load the MNIST dataset using the Keras load_data function
(define dataset (tf.keras.datasets.mnist.load_data))

;; prepare the dataset
(define x_train (/ (get-item (get-item dataset 0) 0) 255.0))
(define y_train (get-item (get-item dataset 0) 1))
(define x_test (/ (get-item (get-item dataset 1) 0) 255.0))
(define y_test (get-item (get-item dataset 1) 1))

(define loss_fn (tf.keras.losses.SparseCategoricalCrossentropy (dict '((from_logits : #t)))))

(let* ([optimizer (choose "adam" "rmsprop" "sgd")]
       [dropout_rate (choose 0.0 0.1 0.2 0.4)]
       [activation (choose "relu" "sigmoid")]
       [hidden_layer_size (choose 10 20 30)]
       [options (dict `((optimizer : ,optimizer)
                        (loss : ,loss_fn)
                        (metrics : ,(vector "accuracy")))))]
       [epochs 5]
       [experiment (comet_ml.Experiment (dict '((project_name : "calysto-scheme"))))]
       [model (tf.keras.models.Sequential
                (vector
                 (tf.keras.layers.Flatten (dict '((input_shape : (28 28)))))
                (tf.keras.layers.Dense hidden_layer_size (dict `((activation : ,activation))))
                   (tf.keras.layers.Dropout dropout_rate)
                   (tf.keras.layers.Dense 10)))])
  (print experiment.url)
  (model.compile options)
  (model.summary)
  (experiment.log_parameters (dict `((optimizer : ,optimizer)
                                     (dropout_rate : ,dropout_rate)
                                     (activation : ,activation)
                                     (hidden_layer_size : ,hidden_layer_size)
                                     (epochs : ,epochs)))
                             (dict))
  (experiment.set_model_graph model)
  (let ([history (model.fit x_train y_train (dict `((epochs : ,epochs))))]
        [step 0])
    (map (lambda (key)
           (set! step 0)
           (map (lambda (v)
                  (experiment.log_metric key v step)
                  (set! step (+ step 1)))
                (get-item history.history key)))
         history.history))
  (experiment.end))
```

Fig. 5. Calysto Scheme code for running Tensorflow experiments.

that provides tools for managing experiments. The `choose` operator is used to select a particular combination of "hyperparameters" for an experiment, and a neural network is then defined using Tensorflow functions. Many Tensorflow functions are configured via keyword parameters in Python, which here we provide as Calysto Scheme dictionaries. After an experiment using a particular set of hyperparameters has run to completion, we can simply invoke `(choose)` to pick a new combination of hyperparameters and rerun the experiment.

6 CALYSTO SCHEME IN JUPYTER

The Jupyter Project is a language-independent client-server system for running code using a web browser as the IDE [7]. Initially, Jupyter was limited to Python (and was originally called IPython [10]). However, the developers realized that their system could be used for any language, and the

Jupyter Project was born. Today, Jupyter is one of the most-used tools in data science. Jupyter Notebooks allow the mixing of code, text, mathematics, and visualizations within a single executable document.

In many ways, the goals of Jupyter were similar to the original goals of Calysto Scheme: language independence with a common UI. In fact, when Jupyter was announced, we realized that this made more sense than our C#-based system, and we began migration of our project to the Jupyter framework. Calysto Scheme, in fact, gets its name from Callisto, the second largest moon of the planet Jupiter.

Running Calysto Scheme in a Jupyter environment (such as notebook, jupyterlab, console, or qtconsole) provides the following additional features:

- TAB completions of Scheme functions and variable names
- Ability to directly display rich media such as images
- Access to so-called magics (`%` meta commands)
- Ability to easily run shell commands (using the "`!` command" format)
- LaTeX-style equations and variables
- Additional Python integration

For example, in Calysto Scheme running in Jupyter, one can create a lambda function with the Greek symbol λ in place of the `lambda` keyword by typing **\lambda** and pressing [TAB] (see Figure 7). Building on top of the Jupyter system makes Calysto Scheme easy to use, and brings it into a modern language environment.

It should be noted that many computer science educators find the use of Jupyter Notebooks problematic. The main objection is that it is easy for a student to create hidden state that creates a non-intuitive experience. For example, one can edit cells that have already been executed, creating dependencies that are no longer valid. In spite of this limitation, Jupyter Notebooks are popular with data scientists and others who tell stories with code, visualizations, and text. For more on this topic, see [9].

Jusging from the Calysto Scheme github reported issues, most people discover Calysto Scheme through the Jupyter Project's "Try Jupyter" page where Calysto Scheme is featured along with kernels in C++, Julia, GNU Octave, R, and Ruby [8]. Many people apparently use Calysto Scheme for working through the SCIP exercises. See Figure 6.

7 STACK-LIKE TRACEBACKS FOR DEBUGGING

In Calysto Scheme, tracebacks are available to help with debugging. For example, in Figure 8 the definition of the factorial function contains a typo, with `q` instead of `1` for the base case, which generates an unbound variable error. The traceback shows the sequence of recursive calls that were made before the error occurred. In many Scheme implementations, since control is not managed with a stack, it is not possible to generate such a stack trace. However, in Calysto Scheme stack tracing can be enabled or disabled via the parameter `use-stack-trace`. For example, calling `(use-stack-trace #f)` turns off stack tracing.

8 SCHEME IN PYTHON, PYTHON IN SCHEME

An approach commonly taken in teaching a course in Programming Languages (PL) is to implement an interpreter for a subset of Scheme in another language, such as Python. This is a relatively straightforward process, which is considerably simplified by Scheme's easy-to-parse syntax. For example, the following code shows part of an interpreter written in Python with the functions

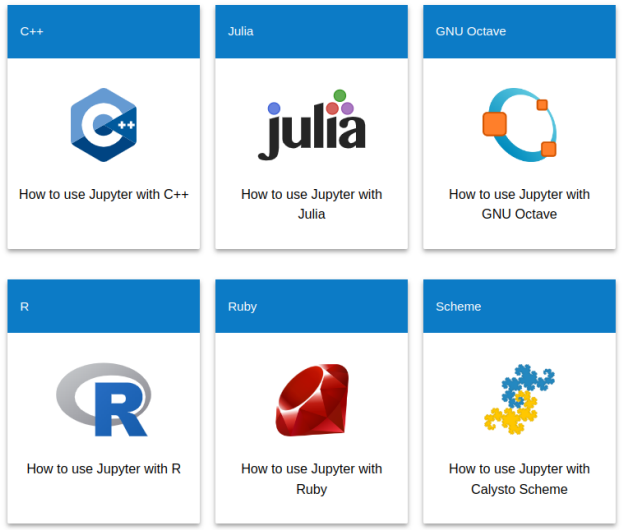

Fig. 6. Calysto Scheme is available online via the Jupyter Project's "Try Jupyter" page.

```
In [1]: (define factorial
          (λ (n)
             (if (= n 0)
                 1
                 (* n (factorial (- n 1))))))

In [2]: (factorial 5)

Out[2]: 120
```

Fig. 7. Calysto Scheme running in a Jupyter notebook.

```
In [3]: (define factorial
          (λ (n)
             (if (= n 0)
                 q
                 (* n (factorial (- n 1))))))

In [4]: (factorial 5)

Traceback (most recent call last):
  File "In [4]", line 1, col 1, in 'factorial'
  File "In [3]", line 5, col 15, in 'factorial'
  File "In [3]", line 5, col 15, in 'factorial'
  File "In [3]", line 5, col 15, in 'factorial'
  File "In [3]", line 5, col 15, in 'factorial'
  File "In [3]", line 5, col 15, in 'factorial'
  File "In [3]", line 4, col 10
RunTimeError: unbound variable 'q'
```

Fig. 8. Example of a traceback in Calysto Scheme.

`evaluator` and `apply_operator` for evaluating Scheme expressions and applying Scheme functions, respectively:

```
# Scheme-in-Python interpreter

# define parser, reader, tokenizer, and utilities (Map, car, cdr, etc.)

def evaluator(expr):
    if car(expr) == "literal":
        return cadr(expr)
    elif car(expr) == "application":
        return apply_operator(evaluator(cadr(expr)),
                              Map(evaluator, caddr(expr)))
    else:
        raise Exception("Invalid AST: %s" % expr)

def apply_operator(op, operands):
    if op == "+":
        return sum(operands)
    else:
        raise Exception("Unknown operator: %s" % op)

evaluator(parser(reader(tokenizer("(+ 1 2)"))))
→ 3
```

Conversely, one could also teach PL principles by implementing Python in Scheme. However, this would be much more difficult to attempt in a single semester course, due to the complexity of Python's syntax. However, if it were possible to outsource the parsing of Python syntax into Abstract Syntax Tree (AST) structures, then one could commence with building a Python interpreter written in Scheme that operates directly on Python ASTs. Because Calysto Scheme can call Python libraries, this becomes a simple task, as the `ast` Python library has the ability to take strings of Python code and turn them into ASTs [2].

Mirroring the above Python functions for interpreting Scheme code, one can easily construct similar functions written in Scheme for interpreting Python ASTs:

```
;; Python-in-Scheme interpreter

(import "ast")

(define evaluator
  (lambda (ast_expr)
   (cond
     [(isinstance ast_expr ast.Module)
      (evaluator (get-item ast_expr.body 0))]
     [(isinstance ast_expr ast.Num)
      ast_expr.n]
     [(isinstance ast_expr ast.Expr)
      (evaluator ast_expr.value)]
     [(isinstance ast_expr ast.BinOp)
      (apply-operator ast_expr.op
         (evaluator ast_expr.left)
         (evaluator ast_expr.right))]
     [else (error 'evaluator (format "Unknown ast: ~s" ast_expr))])))

(define apply-operator
  (lambda (op v1 v2)
    (cond
      [(isinstance op ast.Add) (+ v1 v2)]
      [else (error 'apply-operator (format "Invalid operator: ~s" op))])))

(evaluator (ast.parse "1 + 2"))
→ 3
```

The main differences between the Scheme interpreter in Python, and the Python interpreter in Scheme arise from the different ways in which they represent abstract syntax. For example, the Scheme expression `(+ 1 2)` is treated as a function application AST by the Scheme-in-Python interpreter, whereas the Python expression `"1 + 2"` is treated as a binary operator AST by the Python-in-Scheme interpreter. One could easily change the Scheme abstract syntax structures to more closely mimic the Python AST structures if desired, in order to make the two interpreters more parallel.

9 SUMMARY

Nothing to see here. Move along.

ACKNOWLEDGMENTS

We did it all ourselves.

REFERENCES

[1] Harold Abelson and Gerald Sussman with Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press/McGraw-Hill, Cambridge, MA.

[2] Douglas Blank. 2016. *Jupyter Notebook: Implementing Python in Scheme.* Bryn Mawr College. Retrieved June 30, 2023 from https://jupyter.brynmawr.edu/services/public/dblank/CS245%20Programming%20Languages/2016-Fall/Notebooks/PythonInScheme2.ipynb

[3] Douglas Blank, Jennifer S. Kay, James B. Marshall, Keith O'Hara, and Mark Russo. 2012. Calico: a multi-programming-language, multi-context framework designed for computer science education. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE 2012).* ACM, New York, 63–68.

[4] Daniel P. Friedman and Mitchell Wand. 2008. *Essentials of Programming Languages* (3rd ed.). MIT Press, Cambridge, MA.
[5] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. 1999. Trampolined style. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*. ACM, New York, 18–27. https://doi.org/10.1145/317636.317779
[6] Mark Guzdial. 2003. A media computation course for non-majors. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '03)*. ACM, New York, 104–108. https://doi.org/10.1145/961511.961542
[7] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.
[8] The Jupyter Project Maintainers. 2023. . The Jupyter Project. Retrieved July 14, 2023 from https://jupyter.org/try
[9] Keith O'Hara, Douglas Blank, and James Marshall. 2015. Computational notebooks for AI education. In *Proceedings of the 28th International FLAIRS Conference*. AAAI Press, Palo Alto, CA, 263–268. https://doi.org/10.13140/2.1.2434.5928
[10] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53
[11] Llewellyn Pritchard. 2008. . Retrieved July 13, 2023 from https://xacc.wordpress.com/2008/11/03/ironscheme-does-cps/
[12] Llewellyn Pritchard. 2022. . Retrieved July 14, 2023 from https://github.com/IronScheme/IronScheme
[13] Calysto Scheme Project. 2023. . GitHub. Retrieved June 30, 2023 from https://github.com/Calysto/calysto_scheme
[14] IronRuby Project. 2011. . .NET Foundation. Retrieved June 30, 2023 from http://ironruby.net
[15] Jay Summet, Deepak Kumar, Keith O'Hara, Daniel Walker, Lijun Ni, Douglas Blank, and Tucker Balch. 2009. Personalizing CS1 with robots. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education (SIGCSE 2009)*. ACM, New York, 433–437.

Received 14 July 2023; revised .....; accepted .....