PScheme: A Pascal Powered Scheme

Language Specifications

Joshua Fredricks

February 19, 2019

1 Overview

PScheme is a functional, weakly-typed, and interpreted programming language with a simplistic design philosophy based on the Scheme family of programming languages. PScheme is implemented in Pascal which is then compiled using the Free Pascal Compiler.

Due to its simplistic nature, only four data types are supported, which include integers, booleans, cons cell, and nil, which represents the empty list.

When data values are stored in memory, the contents of said object is dynamically allocated. Because Pascal only supports manual garbage collection, the deallocation of memory is performed manually in PScheme.

In PScheme, all objects are first-class values. That is to say, such values can be passed as arguments to procedures, returned as values from procedures, and can be stored in data structures and combined to form new objects.

PScheme variables and keywords are lexically scoped. That is to say, identifiers may be bound globally, at top level, or locally, within a list or a block of code. Specifically, local bindings are only visible within the corresponding block of code and any identifiers that share the same name but exists outside of said block refers to a different binding. Likewise, for an identifier without a top level binding that shares the same name as the local binding, the reference is invalid. The scope of a binding is considered to be the block in which the bound identifier is visible.

Procedures are first-class data objects similar to other data objects such as integers and booleans. Similarly, variables are bound to procedures in the same manner, without distinction. Moreover, when a procedure is defined, all lexical content is also carried over when the said procedure is invoked, as per lexical scoping.

The following sections will discuss the syntax and structure of PScheme along with examples.

2 Syntax and Structure

2.1 Identifiers, Data Types, and Notation

Identifiers, which include variables and keywords, may consist of a combination of ASCII characters that correspond to the hexadecimal codes $21 - 7E$. When naming an identifier, digits and arithmetic operators may not be used.

The integer data type is limited to 32-bit signed integer range. The boolean data values *true* and $false$ are represented as $\#t$ and $\#f$, respectively. The *nil* data type represents the empty list. The cons cell abstract data type will be discussed later.

Structured forms and lists are enclosed within parentheses. In order to call the value of a variable reference, parentheses are not used. Entering the variable reference will return the value unless the the variable is unbounded, in which case an error will occur.

Operations are represented in prefix notation, such that the operator is stated first followed by the operands, generally following the form:

$$(operator\ operand_1\ operand_2\ ...\ operand_n)$$

For any operation of n-operands.

2.2 Arithmetic Operations

PScheme supports the basic arithmetic operators, including addition ((+)), subtraction ((−)), multiplication ((∗)),division ((/)), and modulo ((*mod*)). Arithmetic operators can receive arguments and return values in the form of 32-bit signed integers.

Examples:

$> (+ 3\ 2)$
5
$> (- 4\ 5\ (/\ 10\ 5))$
-3

2.3 Binding Variables

The keyword (*define*) is used to bind variables and/or expressions to values or parameterized expressions in the form of polymorphic or monomorphic procedures. In general, all values in PScheme are mutable. When binding variables to values or procedures, a single namespace is used.

Examples:

$> (define\ x1\ (+\ 5\ 2))$ *Monomorphic procedure*
$> x1$
7
$> (define\ (foo\ n)$
$\quad (+\ n\ (foo\ (-\ n\ 1))))$ *Polymorphic procedure defined recursively*
$> (foo\ 3)$
6

2.4 Cons Cell

The cons cell is a abstract data type consisting of two parts, a constructor, denoted by (*cons*), and selectors, denoted by (*cdr*) and (*car*). The (*cons*) procedure constructs cons cells, or pairs, and may take two arguments of any data type to construct said pair. A sequence of pairs and concomitant calling of (*cons*) creates lists. The first value of the cons cell is called the (*car*), while the second or remaining values in a pair or list, respectively, is called the (*cdr*). When called, the (*car*) procedure returns the *car* of the cons cell it is applied to. Likewise, the (*cdr*) procedure retuens the *cdr* of said cons cell. The pair and list can be visualised as such:

$$[(car)\ |\ (cdr)] \quad Representation\ of\ (car)\ and\ (cdr)$$
$$[1\ |\ 2] \quad Cons\ Cell/Pair$$
$$[1\ |\ ] \longrightarrow [2\ |\ ] \longrightarrow [3\ |\ 4] \quad List$$

Lists are considered either proper or improper. In a proper list, the value of (*cdr*) in the last cons cell is (*nil*). On the other hand, an improper list may have any value other than (*nil*) in the (*cdr*) cell of the last cons cell. The resulting list or pair created with the (*cons*) procedure will be returned in Dotted Pair Notation (DPN). Because PScheme does not have a symbol or quote data types, lists may not be an argument for the (*cons*) procedure, however, lists stored in a variable may be used as an argument.

Examples:

$>$ $(cons\ 1\ (cons\ 2\ (cons\ 3\ nil)))$

$(1\ .\ (2\ .\ (3\ .\ nil)))$ *Proper list in DPN*

$>$ $(define\ foo\ (cons\ 1\ (cons\ 2\ (cons\ 3\ nil))))$

$>$ $(define\ bar\ (cons\ 1\ (cons\ 2\ (cons\ 3\ 4))))$

$>$ $(car\ bar)$

1

$>$ $(cdr\ foo)$

$(2\ .\ (3\ .\ (4\ .\ nil)))$

$>$ $(define\ baz\ (cons\ foo\ bar))$

$>$ baz

$(1\ .\ (2\ .\ (3\ .\ (nil\ .\ (1\ .\ (2\ .\ (3\ .\ 4)))))))$

$>$ $(car\ (cdr\ (cdr\ baz)))$

3

2.5 Local Bindings

The (let) procedure is used to evaluates an expression using a locally bounded expression. The locally bounded expression is sequentially evaluated, hence, the value of the locally bounded expression is also used for any locally bounded expressions following it. That is to say, the scope of the locally bounded expression begins after the declaration of the expression and extends to the end of the block.

Examples

$>$ $(let\ ((foo2))\ (+\ foo\ 6)))$

8

$>$ $(let\ ((bar\ 2)\ (baz\ (+\ bar\ bar)))\ (+\ bar\ baz))$

6

2.6 Predicates

The procedures $(int?)$ and $(bool?)$ receives an argument of any data type and returns $\#t$ if the argument is an integer and boolean, respectively. Otherwise, the procedures will return $\#f$ if the argument is not an integer and not a boolean, respectively. The $(eq?)$ procedure checks if the given arguments are physically equivalent (holds the same address in memory). If the arguments are physically equivalent, $\#t$ is returned, otherwise $\#f$ is returned. PScheme also supports inequality symbols ($<$, $>$, $=$, $<=$, $>=$) that compare two integers and returns $\#t$ or $\#f$ based on the result of the operation.

Examples:

$>$ $(bool?\ (<\ 5\ 3))$

$\#t$

$>$ $(<\ 5\ 3)$

$\#f$

$>$ $(int?\ (+\ 3\ 6))$

$\#t$

$>$ $(eq?\ (cons\ 1\ 2)\ (cons\ 1\ 2))$

$\#f$

$>$ $(define\ a\ (cons\ 1\ 2))$

$>$ $(eq?\ a\ a)$

$\#t$

2.7 Conditional and Short-Circuit Evaluation

The (if) statement is of the form $(if\ condition\ consequent\ alternative)$ and takes a condition, consequent, and alternative as arguments. If the condition returns $\#t$ then the consequent is evaluated,

otherwise, of the the condition returns $\#f$ then the alternative is evaluated.

The (and) procedure is evaluated from left to right and can take an n-number of operands as arguments. $\#f$ is returned when the first operand to return $\#f$ is evaluated, the remaining operands are left unevaluated. If all operands return $\#t$ then the value of the last operand is returned. If there are no operands then $\#t$ is returned.

Similar to the (and) procedure, the (or) procedure is also evaluated from left to right and may take an n-number of operands as arguments. If all of the argument values return $\#f$ then $\#f$ will be returned by the procedure. The value of the first operand to be evaluated as $\#t$ will be returned, and the remaining operands are left unevaluated. If there are no operands, the procedure returns $\#f$.

Examples:

$> (and\ 5\ (+\ 3\ 5)\ (<\ 5\ 1)\ \#t)$

$\#f$

$> (or\ (int?\ \#t)\ 15\ (=\ 6\ 3))$

15

$> (+\ (and\ 1\ 2\ 3)\ (or\ 4\ 5\ 6))$

7

$> (define\ foo\ 2)$

$> (if\ (or\ (<\ 3\ foo)\ (bool?\ foo)\ foo)\ (*\ foo\ foo)\ (-\ foo\ 5))$

4

2.8 Garbage Collection

When variables are bounded, the values of said variable is dynamically allocated. We can deallocate a variable by using the procedure ($free$).

Example:

$> (define\ foo\ 5)$

$> foo$

5

$> (free\ foo)$

$> foo$

$> Error :< Unbound\ Variable >$

2.9 Errors

When an error occurs in PScheme, an error message in the form of $Error :< Type\ of\ error >$ will appear. Error checking occurs during runtime and, thus, the error message will appear when the error is encountered during the running of the program. Errors may include improper use of procedures, unbounded identifiers, integers outside the range, missing or extra parentheses, type errors, and invalid syntax.

2.10 Example Program

The following question comes from Problem 48 on the ProjectEuler.net website:

> The series, $1^1 + 2^2 + 3^3 + ... + 10^{10} = 10405071317$.
> Find the last ten digits of the series, $1^1 + 2^2 + 3^3 + ... + 1000^{1000}$.

Solution:

```
(define (powerof num hat)
    (if (= hat 0)
        1
        (* num (powerof num (- hat 1)))
    )
)

(define (sumofseries n)
    (if (= n 1)
        1
        (+ (powerof n n) (sumofseries (- n 1)))
    )
)

(mod (sumofseries 1000) 10000000000)
```

9110846700

References

[1] Dybvig, R. K. (2003). The Scheme Programming Language (Third ed.). Cambridge: The MIT Press.

[2] Hughes, C. (n.d.). Retrieved February 19, 2019, from https://projecteuler.net/problem=48

[3] Pearce, J. (1998). Programming and Meta-Programming in Scheme (Undergraduate Texts in Computer Science). New York, NY: Springer.