# SX DOCUMENT

**Stewart A. Brown**
**Dennis Braddy**

## Disclaimer

## The PACT SCHEME Language



## The PACT SCHEME/SX Reference

## Interfacing Compiled and Interpreted Code



## Related Documents

## Acknowledgements

A special acknowledgement is due to Lee Taylor who designed and implemented the internal interface between the PACT SCHEME interpreter and PDBLib. That code eventually became SX when the remainder of PACT was thrown into the pot.

# Disclaimer



Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under

Contract W-7405-Eng-48.

# The PACT SCHEME Language

This section introduces the reader to the PACT implementation of the SCHEME programming language. It does this through a formal description of the language (section 1.2) and a tutorial (section 1.3). For readers who are already familiar with SCHEME in general, this section may be skimmed. Note should be taken, however, of the differences between PACT SCHEME and other SCHEME implementations.

## Introduction

### SCHEME

The LISP programming language is one of the most powerful computer languages ever developed. Somewhat ironically, it is also one of the simplest to implement, at least to a certain level. Many applications would benefit by being able to provide their users with the ability to extend the application via a macro or scripting mechanism. LISP turns out to be ideal for this purpose. The language is syntactically simple and can be molded to meet a variety of needs. It also offers more power than most applications care to provide to their users, but one would rather be in that position than in having an underpowered extension mechanism.

Scheme is a lexically scoped, properly tail recursive dialect of the LISP programming language. The PACT implementation is described abstractly in Abelson and Sussman's book, *Structure and Interpretation of Computer Programs*. It features all of the "essential procedures" described in the *Revised$^3$ Report on Scheme* (Jonathan Rees and William Clinger, ed) which defines the standard for Scheme.

PACT SCHEME is implemented as a library; however, a small driver delivers a stand alone SCHEME interpreter. This is useful for people who want to learn the language as well as people who want to prototype algorithms (one of LISP's strong points).

The PACT implementation features a reference counting incremental garbage collector. This distributes the overhead of memory management throughout the running of Scheme code. It also tends to keep Scheme from trying to grab the entire machine on which it is running which some garbage collection schemes will attempt to do.

### SX

SX is perhaps the ultimate PACT statement. It is simply Scheme plus the other parts of PACT. A more precise way to describe it is as a dialect of LISP with extensions for PGS, PDB, PDBX, PML, and PANACEA. What this yields is an interpretive language whose primitive procedures span the functionality of all of PACT.

Like the PACT SCHEME implementation which it extends, SX provides both a library and a stand alone application. The stand alone interpreter is the engine behind applications such as PDBView and PDBDiff. These examples demonstrate the extreme power of the PACT design. The modularization and layering make it possible to use the PACT components like building blocks.

In addition to the above mentioned functionality, SX contains functionality which is the generalization of that found in ULTRA II. This means that as the development of SX proceeds, an SX driven application will be able to perform arbitrary dimensional presentation, analysis, and manipulation tasks.

Because of the fundamental unity of these two PACT parts, they are documented in a single manual. The first part will cover the "standard" Scheme functionality and the second part will discuss the SX extensions.

# SCHEME Language Guide

The PACT SCHEME implementation derives from two main sources. First, Abelson and Sussman's book, *The Structure and Interpretation of Computer Programs* gives an account of a basic Scheme interpreter. To fill in the details and build a somewhat standard Scheme, I used the *Revised$^3$ Report on the Algorithmic Language Scheme*. This has subsequently been updated with some of the features of the *Revised$^4$ Report on the Algorithmic Language Scheme* (R4RS).

This implementation differs from standard Scheme in a few fundamental ways. First, the numerics defined in R4RS are not suited to the main audience of PACT. For example, the notions of exactness and inexactness of floating point numbers are not practical concepts in actual numerical simulation codes. Furthermore, for ease of implementation and good matching to the needs of applications using the library version of the interpreter, the numeric types implemented map to the standard types of the C language. In short, I implemented integers, reals, and characters as C longs, doubles, and chars respectively. Second, keeping in mind that this interpreter is implemented in C and needs to be as compatible with C usages as possible, I chose to follow C conventions in situations where it was the convenient and "obvious" thing to do.

Throughout this manual little will be said of the differences between Scheme and LISP in general. The reader may assume that unless otherwise explicitly stated comments about the character of Scheme apply also to LISP in general. The functions in Scheme differ in many cases from those in LISP but the character of the languages is similar. For example, they are both fully parenthesized and they both use prefix notation.

## General Concepts of Scheme

The term object is used to refer to an instance of any type of Scheme variable, data, or expression. This usage is similar to that used in connection with other programming languages. One of the distinguishing characteristics of Scheme is that every object in Scheme has a definite (although sometimes only implicit) type which can be queried. For example, an object, x, can be asked if it is a real number

```
(real? x)
```

All objects in Scheme have an internal representation and printing representation. That is to say that every object that is represented internally can be indicated in printed form in output operations. Most objects can be indicated in input operations by their printing representation. For example, the string "foo bar baz" has some internal representation in Scheme and has a printing representation that can be used in input or appears in output as

```
"foo bar baz"
```

By contrast, an object such as a hash table has an internal representation and a printing representation, but the printing representation cannot be used in input operations to indicate that a complete hash table should be inferred from the printing representation. The printing representation might look like

```
<HASH_TABLE|521>
```

This clearly does not contain enough information to infer the entire contents of the hash table whose internal representation it is associated with.

The most important data structure in LISP is the list (LISP stands for LISt Processing language or, if you choose to believe, Lots of Irritating Silly Parentheses). This is also true of Scheme. Both programs and data are represented by lists. This feature of the language permits Scheme programs to manipulate Scheme programs as data. This is the source of a great deal of Scheme's power.

Scheme is a tail recursive dialect of LISP. This means that certain recursively posed procedures execute as simple loops. This permits the Scheme programmer to write procedures using recursion and the interpreter (or compiler) to run these procedures in fixed space. The function call stack is never blown with tail recursive procedures.

Variables in Scheme are said to be bound when they are assigned to locations in which a value may be stored. Variables in Scheme may be either bound, in which case references to the variable yield a value, or unbound, in which case references to the variable yield an UNBOUND VARIABLE error.

The collection of all variables visible at any point in the execution of a program is called the environment in effect at that point. When interacting directly with the Scheme interpreter at the top level (no procedures executing) the environment in effect is called the global environment.

Scheme like C is a call by value language.

Scheme has a class of procedures called predicates. Predicate procedures return one of the two boolean values: TRUE or FALSE (represented by #t or #f respectively). Most languages have syntactic constructs to express this functionality. In Scheme which has a simple and uniform syntax (except for quote and quasi-quote) this functionality must be expressed as procedures.

## Documentation Conventions

In this manual the following notational conventions will apply.

- => when used in connection with a Scheme expression means "evaluates to". This will be used to show what the interpreter returns when it evaluates an expression.
- the terms expression and form will be used interchangeably.

## Syntax and Notation

It is a characteristic of LISP dialects to use prefix notation, that is operators precede their operands. LISP and its dialects also use parentheses to group and associate objects. Parentheses are the only syntactic mechanisms for separating groups of objects. So to add four numbers, 3, 25.2, 0.015, and -1.3e-4, one would use the expression

```
(+ 3 25.2 0.015 -1.3e-4)
```

Part of the power of the language derives from the fact that such expressions return values that can be used in other expressions, for example

```
(* 5 (+ 3 25.2 0.015 -1.3e-4))
```

which multiplies the result of adding 3, 25.2, 0.015, and -1.3e-4 by 5.

### Whitespace

The characters (expressed in C style notation) ‘ ’ (blank),’\t’ (tab), ‘\f’ (formfeed), and ‘\n’ (newline) are considered to be whitespace in Scheme programs.

### Comments

A semicolon begins a comment, and the comment extends to the end of the line containing the semicolon.

### Numbers

Numbers begin with any of the following characters: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘.’, or ‘-’. They must correspond to the standard representations of either integer or floating point types used in C (and most FORTRANs).

### Strings

Strings are arbitrary groups of characters which are delimited with double quotes ‘"’.

### Identifiers

Subject to the special case listed below, all groups of characters delimited by whitespace which are neither numbers nor strings are identifiers. Case is significant in PACT Scheme unlike R4RS.

**Special Syntax**

The following characters have special meanings in some contexts. They should not be used in identifiers unless escaped by ‘\’.

| | |
|---|---|
| . | used in the notation of pairs and dotted argument lists |
| ( | begins a list of objects |
| ) | ends a list of objects |
| ’ | quote (don’t evaluate) an object |
| ‘ | quasi-quote an object |
| , | used in conjunction with quasi-quotation |
| ,@ | used in conjunction with quasi-quotation |
| # | used with character constants, boolean constants, and vector constants |
| \ | used with character constants and in strings as an escape character |

**Naming Conventions**

Predicate functions (returning true or false) end with ‘?’.

Functions which store values end with ‘!’.

Functions which map objects of different types have "->" in their names. For example, the function which maps a list into a vector is called list->vector.

**Reserved Identifiers**

Identifiers typically occur as the names of variables (or symbols) or functions (or procedures). PACT Scheme allows programmers to use identifiers that have been assigned to the built in procedures of the language for their own variables and functions, but this has the effect of hiding (possibly for the duration of the session or program) the built in procedure. It is recommended that this not be done in the case of the following built in functions. They can be thought of as reserved keywords.

| | | |
|---|---|---|
| and | if | quasiquote |
| begin | lambda | quote |
| define | let | set! |
| do | let* | unquote |
| else | or | unquote-splicing |

## Scheme Data Types

The fundamental data types of PACT Scheme are:

| | |
|---|---|
| **boolean** | there are two boolean objects in Scheme, #t and #f, representing a TRUE and FALSE logical value respectively |
| **char** | a character object is a single character and is represented by its printing representation preceded by a #\, e.g. #\A represents 'A' |
| **number** | a number is any object with a numeric type such as integer or real |
| **pair** | an object with two parts called the car and cdr. This is the primitive object from which lists are built |
| **string** | a collection of characters (not character objects) delimited by double quotes. |
| **symbol** | a synonym for variable |
| **procedure** | an object which specifies a procedure |
| **vector** | a 1 dimensional array of objects |
| **guest** | an object that encapsulates C variables and structures as a part of the interface between PACT Scheme and C applications |

Each of these types has a corresponding predicate function. It is guaranteed that no Scheme object will return TRUE for more than one of the associated predicates.

In addition the number type is subdivided into

| | |
|---|---|
| **integer** | an object representing an integer number |
| **real** | an object representing a floating point number |

PACT Scheme does not implement

**rational**

**complex**

The reason for this is that in order to keep the implementation reasonably small and efficient, I chose to adhere closely to underlying C concepts and data types. Both of these types could be implemented as C structures, but handling the arithmetic with them is not consistent with the goals of the implementation.

Predicates exist for the integer and real types, but any object that is either an integer or a real is also a number.

## Scheme Expressions

An expression in Scheme is an object which when evaluated returns a value. Scheme has a small set of primitive expression types and all other expressions in Scheme can be built up or derived from more primitive expressions. The primitive expression types are enumerated in this section.

This is the heart of Scheme, and although this is not by any means a tutorial, once the concept of expression is mastered along with the primitive expression types, the remainder of the work of this manual is filling in details on the set of procedures supplied in PACT Scheme.

**Literal Expressions**

An expression consisting of a constant or a quoted expression is a literal expression. Literal expressions evaluate to themselves. That is a string evaluates to itself, a number evaluates to itself, and so on.

Another kind of literal expression comes about by specifying that an expression should not be evaluated. This is a quoted expression. Its syntax is:

    (quote expression)

or

    ‘expression

**Variable Definition**

Variables are unbound until they are defined in a variable definition expression. The syntax is:

    (define variable value)

This expression binds variable to a location where the value is then placed.

**Variable Reference**

An expression representing a reference to a variable consists of the variable itself.

    variable

It is an error to reference an unbound variable, i.e. a variable which has not be been bound in a variable definition expression.

**Variable Assignment**

The value in the location to which a variable is bound can be changed by the use of a variable assignment expression.

    (set! variable value)

It is an error to try to assign a value to an unbound variable.

**Conditional Expressions**

In Scheme there is a single primitive conditional expression form from which all others may be derived.

```
(if test consequent alternate)
```

If test is not FALSE (evaluates to anything but #f) the consequent expression is evaluated and returned. Otherwise the alternate expression is evaluated and returned. The alternate expression is optional in which case if test is FALSE nothing is evaluated and the conditional expression returns #f.

**Procedure Calls**

Since procedures are a fundamental "data" type in Scheme, the problem of procedure calls splits into two parts. The first part is obtaining a procedure. The next section explores that issue in more detail. The second part is invoking a procedure once you have one. A procedure call is posed as a list consisting of the procedure object as the first element of the list and the arguments to the procedure as the remaining elements of the list.

```
(procedure argument1 ...)
```

Some examples of procedure calls are:

```
(* 2 3 4) => 24

((if #t + -) 20 10 5) => 5
```

In the first example the procedure object is "*" and the procedure call expression multiplies the numbers 2, 3, and 4 to obtain the value 24. In the second example, the conditional expression

```
(if #f + -) => -
```

is evaluated to return the procedure "-" which is then applied to the numbers 20, 10, and 5 to yield the result 5.

**Anonymous Procedures**

A characteristic of LISP which Scheme shares is the notion of an anonymous procedure. An expression called a lambda expression evaluates to a procedure. A lambda expression is the irreducibly primitive procedure definition mechanism for Scheme. All procedures and most of the special expressions except those mentioned in this section can be defined in terms of lambda expressions.

The syntax of a lambda expression is

(lambda arguments body)

The arguments are a list of formal parameters to the procedure, and the body is the set of expressions which define the action of the procedure.

The arguments can be one of the following:

variable

(variable1 ...)

(variable1 ... variablen . variablen+1)

In the last case variablen+1 is bound to a list of the arguments in the procedure call which remain after the first n arguments are bound to the first n formal parameters.

Some examples of lambda expressions in procedure calls are:

```
((lambda x x) 1 "foo" 2.3) => (1 "foo" 2 3)

((lambda (x) (+ 1 x)) 4) => 5

((lambda (x y . z) z) 3 4 5 6) => (5 6)
```

An anonymous procedure can be bound to a variable in order to give a more convenient way of using procedure objects.

```
(define foo (lambda x x))
```

This would bind the variable foo to the lambda expression for the identity function. It can then be used as

```
(foo 3) => 3
```

This usage is so common that an alternate form of the define expression was created. Its usage is

(define (variable arguments) body)

Here the variable is bound to the procedure being made. The arguments are like those of the lambda expression.

none

variable1 ...

variable1 ... variablen . variablen+1

Note the absence of parentheses here. Compare with the lambda expression syntax. The first variant means that there are no arguments.

The body of the procedure is identical with that for the lambda expression.

    (define (foo x) x)

## Scheme Procedures

In most LISP dialects there are two kinds of procedures: those which evaluate their arguments before binding them to the formal parameters; and those which do not evaluate their arguments before binding them to the formal parameters. There are also procedures which carry out an additional evaluation of the result of the procedure call.

PACT SCHEME permits all combinations of evaluating arguments and evaluating results. This makes for a great deal of flexibility in the implementation and in the use of the library in applications. More will be said about this later in this manual.

The majority of Scheme functions evaluate their arguments and do not evaluate the result of the call. These will be referred to simply as procedures. All other functions will be referred to as macros (R4RS calls them special forms).

## Scheme Programs

The conceptual structure of a Scheme program differs from that of a C program primarily in the entry into the program. Both will have a collection of function or procedure definitions, and both have a main entry point into the program.

In the case of C there is a special procedure called main which serves as the main entry point into the program. Arguments from the command line invocation of the C program are made available to main. The main procedure may call no other functions and simply do the work of the program itself or it may do nothing but call other functions.

In Scheme, their is no special function like main in C. A file is read in and variables and functions are defined sequentially. Functions and variables must be defined before they are invoked or referenced. Consequently the analog of the main entry point into a Scheme program must come at the end of a file or function definition, and it is just an ordinary procedure call.

**Example C code fragment illustrating structure of a C program**

---

```
main()
   {foo(1);
    bar();
    exit(0);
foo(int x)
   {...
```

```
bar()
   {...
```

**Example Scheme code fragment illustrating the structure of the same program**

---

```
(define (foo x) ...)

(define (bar) ...)

(foo 1)

(bar)
```

NOTE: a common complaint about LISP programming is that one has to count parentheses carefully which is a lot of bother. The answer to that problem is to use a text editor which indicates balancing parentheses. One common editor with this feature is GNU Emacs.

# A PACT Scheme Tutorial

## About Those Parentheses

Scheme evaluates expressions:

```
=> 3
3
=> (+ 9 2)
11
=> ; Comments can be added after a semicolon.
   ; Note that => is the SX prompt.
   ; For clarity, responses from SX will be outlined.
   (+ 9 2) ; Scheme uses Polish notation.
11
```

Scheme evaluates expressions from left to right and from inside nested pairs of parentheses out:

```
=> (- (+ 4 2) 7)

-1
```

The first argument (+ 4 2) is evaluated to 6, and the second argument 7 is evaluated to 7.

Then (- 6 7) is evaluated to -1.

A little more integer arithmetic will show how the expression (9 + 2) (7 - 4)/3 might be written in Scheme:

```
=> (+ 9 2)                  ; i.e., 9 + 2
11
=> (- 7 4)                  ; i.e., 7 - 4
3
=> (* (+ 9 2) (- 7 4))      ; i.e., (9 + 2) (7 - 4)
33
=> (/ (* (+ 9 2) (- 7 4)) 3) ; Q.E.D., (9 + 2) (7 - 4)/3
11
```

Thus, (9 + 2) (7 - 4)/3 is evaluated in three steps:

```
(* (+ 9 2) (- 7 4)) 3)  -> (/ (* 11 3) 3)
(/ (* 11 3) 3)          -> (/ 33 3)
(/ 33 3)                -> 11
```

## Lists and Conses

One of the fundamental and powerful data structures of Scheme is a pair of pointers referred to variously as a cons, cell, or pair. To talk about Scheme you have to talk about conses. The printed representation of cons containing a and b is

```
(a . b)
```

The first part of the pair is called the car of the cons and the second part is called the cdr of the cons.

Lists consist of chains of conses. For example:

```
(a . (b . (c . ())))
```

This is really rather tedious, so a more compact notation is used to represent lists

```
(a . (b . c . ()))) <==> (a b c)
```

The rule is that if the cdr of a cons is another cons the ". (" is suppressed. This also means that you will see things like:

```
(a b c . d)
```

which is equivalent to

(a . (b . (c . d)))

Compare this with the above to see another rule about lists and conses. If the final conses cdr is (), then suppress the " . ()". For example:

(a . ()) <==> (a)

The Scheme function cons is used to create a new cons:

=> (cons 1 2)

(1 . 2)

You can always use cons to build up lists:

=> (cons 1 (cons 2 (cons 3 ())))

(1 2 3)

However there is a better way. Use the list function:

=> (list 1 2 3)

(1 2 3)

## Writing Scheme Functions

Scheme provides predefined primitive functions such as sqrt, equal?, and cons:

=> (sqrt 25)

5.0

=> (equal? 3 5)

#f

=> (equal? (* 3 6) (+ 9 9))

#t

=> (cons ‘fish ‘())

(fish)

=> (cons ‘fish1 (cons ‘fish2 (cons ‘fish3 ‘(fish4))))

```
(fish1 fish2 fish3 fish4)
```

The function equal? is one of a number of Boolean functions that return a true-or-false Boolean object: either #t or #f. The input of a decision-making function like cond (see below) is often the output of a Boolean function:

```
=> (cond ((equal? 2 2) "The numbers are equal.")
         (else "The numbers are not equal."))
"The numbers are equal."
=> (cond ((equal? 2 3) "The numbers are equal.")
         (else "The numbers are not equal."))
"The numbers are not equal."
```

In the two examples above, if equal? evaluates to #t, the first message is printed. Otherwise, the second message is printed.

Scheme programming consists of writing your own functions, e.g.:

```
=> (define (net-vol x2 y2 z2 x1 y1 z1)

     (- (* x2 y2 z2) (* x1 y1 z1)))

net-vol
```

The name of the function (net-vol) was declared in the define statement, and the arguments (six, in this example) were listed. Then the algorithm was stated, giving the locations of the arguments in the algorithm. SX returned the name of the function to confirm that the definition has been accepted.

After net-vol has been defined, it is called with its arguments like any other function:

```
=> (net-vol 6 9 27 4 3 1)

1446
```

Once a function has been defined, it can be included in definitions of other functions. If we defined function square as

```
=> (define (square n)

     (* n n))

square
```

we can define function lim-test as:

```
=> (define (lim-test lim n)
        (<= lim (square n)))     ; Second argument less than
                                 ; or equal to first argument?
lim-test

=> (lim-test 9 100)
#f
=> (lim-test 11 100)
#t
```

## Recursion in Scheme Functions

A Scheme function becomes recursive if it calls itself within its own definition. It takes the general form:

> (define (name ...)
>
> ...
>
> ( name ...))

A conditional statement may be placed in the definition before the function calls itself, in order to eventually escape from the loop. Here is a function that uses the if statement, which takes the logical form:

```
(if (condition)  true false)

=> (define (query n)
      (if (zero? n)
        "The number is zero."
        "The number is not zero."))
query
=> (query 0)
"The number is zero."
=> (query 5)
"The number is not zero."
```

Now we'll define **n-fact**, a function that uses recursion to calculate n!:

```
=> (define (n-fact n)
       (if (= 0 n)
           1
           (* n (n-fact (- n 1)))))
n-fact
=> (n-fact 0)
1
```

```
=> (n-fact 1)
1
=> (n-fact 5)
120
```

## Lists, Symbols, and the car, cdr, and cond Functions

Scheme, which is a dialect of LISP (LISt Processor), processes lists!

A list is a set of space-delimited elements enclosed within a pair of left and right parentheses, e.g.:

(a b c d e)

(Look here)

(1 3 5 7 9)

A list can include other lists:

((a b c) d (e f))

List elements that are not lists are called atoms, e.g.:

| | |
|---|---|
| (a b c d e) | contains the atoms a, b, c, d, and e |
| (Look here) | contains the atoms Look and here |
| (1 3 5 7 9) | contains the atoms 1, 3, 5, 7, and 9 |
| ((a b c) d (e f)) | contains the atom d. |
| ((a b c) d (e f)) | contains the atoms a, b, and c. |
| ((a b c) d (e f)) | contains the atoms e and f. |

The Scheme function **symbol?** tests for a symbol:

=>(symbol? ‘(a b c d e))

#f

=>(symbol? ‘a)

#t

An important list, the null list, contains no elements:

()

In recursive functions, the elements in a list are often removed from the list and operated on one at a time until nothing remains but the null list. The **null?** function can test for the null list to provide an exit from a recursive loop. The looping action would produce results similar to the following non-looping sequence:

```
=>(null? `(1 2 3))

#f                        ; Loop again.

=>(null? `(2 3))

#f                        ; Loop again.

=>(null? `(3))

#f                        ; Loop again.

=>(null? `())

#t                        ; Stop.
```

The output of the function **car** is the first element in a list:

=>(car ‘(a b c d e))

a

=> (car ‘((a b c) d (e f)))

(a b c)

The output of the function **cdr** is the list without the first element:

=> (cdr ‘(a b c d e))

(b c d e)

=> (cdr ‘((a b c) d (e f)))

(d (e f))

The **cond** function uses the following logic:

if condition A is true then action 1

else if condition B is true then action 2

else if condition C is true then action 3

...

else action n

The following function uses the **car** and **cdr** functions to determine the truth of each condition in a **cond** function:

```
=>(define (look-see list1)
; The variable "list1" will be the name given to a list that we will
; examine.
     (cond
; If the first element in the list is "d", print message.
            ((equal? (car list1) ‘d)
             "The d is the first element.")
; If the second element (the first of the rest of the list) is "d",
; print message.
            ((equal? (car (cdr list1)) ‘d)
             "The d is the second element.")
; If the third element (the first of the rest of the rest of the list)
; is "d", print message.
            ((equal? (car (cdr (cdr list1))) ‘d)
             "The d is the third element.")
; If the fourth element (the first of the rest of the rest of the rest
; of the list) is "d", print message.
            ((equal? (car (cdr (cdr (cdr list1)))) ‘d)
             "The d is the fourth element.")
; If the fifth element (the first of the rest of the rest of the rest
; of  the rest of the list) is "d", print message.
            ((equal? (car (cdr (cdr (cdr (cdr list1))))) ‘d)
             "The d is the fifth element.")
; All five tests failed; print message.
          (else "The d is not present in the list.")))
look-see
```

Next we’ll define string **list1** (to which we’ll assign the five elements "a b c d e"), and then we’ll execute function **look-see**:

```
=> (define list1

       ‘(a b c d e))

list1

=>(look-see list1)

d-in-fourth-element
```

The following recursive function uses the null list and the **car** and **cdr** functions to add the numbers in a vector (a list whose elements are numbers). If **v-sum** is null, the result is 0. If **v-sum** is non-null, the first element is added to the **cdr** of **v-sum** recursively until the **cdr** of **v-sum** sum is the null list:

```
=>(define (v-sum v)
          (cond
          ((null? v) 0)
```

```
        (else (+ (car v) (v-sum (cdr v))))))
v-sum
```

Variable v is replaced by a vector when function **v-sum** is executed:

```
=> (define    v
              ‘(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
              16 17 18 19 20 21 22 23 24 25 26
              27 28 29 30))
v
=>(v-sum v)
465
```

## A Matrix Example

In this example we create **matrix-one**, a 5-by-4 matrix. Then, using **matrix-one**, we create **matrix-two**, a 5-by-3 matrix whose first column is the product of columns 1 and 2 in **matrix-one**, and whose second and third columns are columns 3 and 4 in **matrix-one**.

In Scheme, the matrix

2 6 87 92

5 23 15 8

2 32 45 1

56 7 29 4

8 48 25 6

may be written as:

```
=>(define matrix-one
         `((2 6 87 92)
         (5 23 15 8)
         (2 32 45 1)
         (56 7 29 4)
         (8 48 25 6)))
matrix-one
```

We can test this by evaluating **matrix-one**:

```
=> matrix-one

((2 6 87 92) (5 23 15 8) (2 32 45 1) (56 7 29 4) (8 48 25 6))
```

The following expression will select column n in any matrix that contains at least n columns:

```
=> (define (col-n n matrix)

       (cond ((null? matrix) '())

             (else (cons (list-ref (car matrix) n)

                         (col-n n (cdr matrix))))))

col-n
```

What's going on here? First, the **list-ref** function, which hasn't been introduced before, selects the nth element in a list, e.g.,

```
=> (list-ref '(101 102 103 104 105) 3)

104
```

Returning to **col-n**, the **cond** function tests matrix to see if it is null. If matrix isn't null, the **list-ref** function selects element n in the current vector (first row of the [current] matrix = "first matrix"), which will be **cons**-ed with the results of subsequent iterations of **col-n** of the **cdr** of matrix. When a new iteration yields a null, the list of elements is **cons**-ed with ().

Let's test column 1 in **matrix-one**:

```
=>(col-n 1 matrix-one)

(2 5 2 56 8)
```

Let's test column 2 in **matrix-one**:

```
=>(col-n 2 matrix-one)

(6 23 32 7 48)
```

Now that we know how to get columns 1 and 2, we can modify **col-n** slightly to get the product of columns 1 and 2:

```
=> (define (m-times-n m n matrix)

             (cond ((null? matrix) ())

                   (else

                    (cons (* (list-ref (car matrix) m)
```

```
                                    (list-ref (car matrix) n))
                           (m-times-n m n (cdr matrix))))))
m-times-n
```

We now have two column parameters, m and n, and now we **cons** the products of the **list-ref** of m and n of the **car** of matrix.

Let’s test **m-times-n** for columns 1 and 2:

=> (m-times-n 1 2 matrix-one)

(12 115 64 392 384)

Now we can get everything we need from **matrix-one**: any column and the product of any two columns. We need: the product of columns 1 and 2; column 3; and column 4. We get these columns as vectors, so first we’ll lay them out as rows in an intermediate matrix named **matrix-temp**:

```
=> (define matrix-temp
             (list (m-times-n 1 2 matrix-one)
                   (col-n 3 matrix-one)
                   (col-n 4 matrix-one)))
matrix-temp
```

We got **matrix-temp** by cons-ing the product of columns 1 and 2, column 3, column 4, and left & right parentheses.

Let’s see **matrix-temp**:

=> matrix-temp

((12 115 64 392 384) (87 15 45 29 25) (92 8 1 4 6))

Finally, we’ll use **col-n** to get columns 1, 2, 3, 4, and 5 from **matrix-temp**, and we’ll **cons** them all together within left and right parentheses to get **matrix-two**, our objective:

```
=> (define matrix-two
             (list (col-n 1 matrix-temp)
                   (col-n 2 matrix-temp)
                   (col-n 3 matrix-temp)
```

```
                    (col-n 4 matrix-temp)
                    (col-n 5 matrix-temp)))
    matrix-two
```

Now, let’s see **matrix-two**:

=> matrix-two

((12 87 92) (115 15 8) (64 45 1) (392 29 4) (384 25 6))

# The PACT SCHEME/SX Reference

This section constitutes a more formal reference for using PACT SCHEME and SX. There are two main sections (one for SCHEME and one for SX) each of which is divided into two sections (SCHEME level use and C level use).

## PACT Scheme Procedures

Although many of the procedures of Scheme can be expressed in terms of an irreducible set of primitive Scheme functions, the PACT Scheme implementation (as well as the R4RS description) provides a fairly rich set of Scheme functions which are coded at the C level. These run faster by virtue of running from compiled rather than interpreted code, and they make life a little bit easier for programmers by providing many procedures which would otherwise have to be supplied by the programmer.

In PACT Scheme, I implemented two sets of functions. The first set constitutes the core of the interpreter. The second set are not so essential to the majority of applications and on small memory machines are expendable given the choice between having these functions and not fitting on the machine. This second set will be referred to as the large set and the first set will be referred to as the core set. I will document these two sets separately.

The description of the functions consists of the following information:

**name**

Information for C based applications wanting to use this function

Description of the function

Usage of the function as a formal description

Examples

For example:

**if**

C Information: SS_ident, SIF, SS_args

| | |
|---|---|
| Macro: | Evaluate and return the consequent expression if the test evaluates to #t and evaluate and return the alternate expression otherwise. |
| Usage: | (if test consequent alternate). |
| Example: | (if (= i 3) (display "yes") (display "no")) |

## The Core Set

**+**
*C Information:* SS_plus, SS_nargs
*Procedure:* Return sum of args or 0 if no args are supplied.

**-**
*C Information:* SS_minus, SS_nargs
*Procedure:* Return difference of args.

*****
*C Information:* SS_times, SS_nargs
*Procedure:* Return product of args or 1 if no args are supplied.

**/**
*C Information:* SS_divide, SS_nargs
*Procedure:* Return quotient of args (left associative).

**<**
*C Information:* SS_lt, SS_nargs
*Procedure:* Return #t iff the first argument is less than the second.

**<=**
*C Information:* SS_le, SS_nargs
*Procedure:* Return #t iff the first argument is less than or equal to the second.

**=**
*C Information:* SS_neq, SS_nargs
*Procedure:* Return #t iff the first argument is equal to the second.

**>**
*C Information:* SS_gt, SS_nargs
*Procedure:* Return #t iff the first argument is greater than the second.

**>=**
*C Information:* SS_ge, SS_nargs
*Procedure:* Return #t iff the first argument is greater than or equal to the second.

**abs**
*C Information:* SS_abs, SS_nargs
*Procedure:* Return the absolute value of a numeric object.

**acos**
*C Information:* SS_acos, SS_nargs
*Procedure:* Return the arc cosine of the argument.

**and**
*C Information:* SS_ident, SAND, SS_sargs
*Macro:* Evaluate expressions from left to right until one evaluates to #f. The value of the last evaluated expression is returned.

> *Usage:* (and expression$_1$ ... expression$_n$)
>
> *Example:* (and (< 3 4) (list 1 2 3) (< 5 1) (display 5)) => #f

**append**
*C Information:* SS_append, SS_nargs
*Procedure:* Return a new list made from a copy of the first and the second arguments.

**apply**
*C Information:* SS_ident, EE_MACRO, SS_sargs
*Macro:* Cons the procedure onto the args and evaluate the result.

**apropos**
*C Information:* SS_apropos, SS_nargs
*Procedure:* Search the symbol table for documentation.

**ascii-file?**

*C Information:* SS_ascii_filep, SS_sargs
*Procedure:* Return #t if the object is an ascii file and #f otherwise.

**asin**
*C Information:* SS_asin, SS_nargs
*Procedure:* Return the arc sine of the argument.

**assoc**
*C Information:* SS_assoc, SS_nargs
*Procedure:* Return the first list whose car is equal? to the first arg.

**assq**
*C Information:* SS_assq, SS_nargs
*Procedure:* Return the first list whose car is eq? to the first arg.

**assv**
*C Information:* SS_assv, SS_nargs
*Procedure:* Return the first list whose car is eqv? to the first arg.

**atan**
*C Information:* SS_atan, SS_nargs
*Procedure:* Return the arc tangent of the argument.

**begin**
*C Information:* SS_ident, BEGIN, SS_sargs
*Macro:* Evaluate a list of expressions and return the value of the last one.

**boolean?**
*C Information:* SS_boolp, SS_sargs
*Procedure:* Return #t if the object is a boolean and #f otherwise.

**break**
*C Information:* SS_break, SS_zargs
*Procedure:* Enter a Scheme break, return with **return-level**. A Scheme break is simply a recursively executed **read-eval-print** loop. It constitutes a separate loop n which to perform tasks before returning to the preceding evaluation.

> *Usage:* (break)
>
> *Example:* (break)

**caaar**
*C Information:* SS_caaar, SS_sargs
*Procedure:* Return the caaar of the argument.

**caadr**
*C Information:* SS_caadr, SS_sargs
*Procedure:* Return the caadr of the argument.

**caar**
*C Information:* SS_caar, SS_sargs
*Procedure:* Return the caar, i.e. the car of the car, of the argument.

**cadar**
*C Information:* SS_cadar, SS_sargs
*Procedure:* Return the cadar, i.e. the car of the cdr of the car, of the argument.

**caddr**
*C Information:* SS_caddr, SS_sargs
*Procedure:* Return the caddr, i.e. the car of the cdr of the cdr, of the argument.

**cadr**
*C Information:* SS_cadr, SS_sargs
*Procedure:* Return the cadr, i.e. the car of the cdr, of the argument.

**car**
*C Information:* SS_car, SS_sargs
*Procedure:* Return the first member of the pair which is the argument. For a simple pair this is the left hand member and for a list it is the leftmost member of the list.

> *Usage:* (car pair)
>
> *Example:* (car ‘(3 . 4)) => 3

```
(car ‘(1 2 3)) => 1
```

**call-with-cc**
*C Information:* SS_catch, SS_sargs
*Procedure:* Pass an escape procedure (a special procedure of one argument) to the argument.

**cdar**
*C Information:* SS_cdar, SS_sargs
*Procedure:* Return the cdar, i.e. the cdr of the car, of the argument.

**cdaar**
*C Information:* SS_cdaar, SS_sargs
*Procedure:* Return the cdaar, i.e. the cdr of the car of the car, of the argument.

**cdadr**
*C Information:* SS_cdadr, SS_sargs
*Procedure:* Return the cdadr, i.e. the cdr of the car of the cdr, of the argument.

**cddar**
*C Information:* SS_cddar, SS_sargs
*Procedure:* Return the cddar, i.e. the cdr of the cdr of the car, of the argument.

**cdddr**
*C Information:* SS_cdddr, SS_sargs
*Procedure:* Return the cdddr, i.e. the cdr of the cdr of the cdr of the argument.

**cddr**
*C Information:* SS_cddr, SS_sargs
*Procedure:* Return the cddr, i.e, the cdr of the cdr, of the argument.

**cdr**
*C Information:* SS_cdr, SS_sargs
*Procedure:* Return the second member of the pair which is the argument.

*Usage:* (cdr pair)

*Example:* (cdr ‘(3 . 4)) => 4

```
(cdr '(1 2 3)) => (2 3)
```

**ceiling**
*C Information:* SS_ceil, SS_nargs
*Procedure:* Return the smallest integer greater than the argument.

**close-input-file**
*C Information:* SS_cls_in, SS_sargs

*Procedure:* Close the specified input port and release the **IN_PORT** object.

**close-output-file**
*C Information:* SS_cls_out, SS_sargs
*Procedure:* Close the specified output port and release the **OUT_PORT** object.

**cond**
*C Information:* SS_ident, SCOND, SS_sargs
*Macro:* Evaluate the list of cond clauses returning the value of the first one whose first expression does not evaluate to #f. A cond clause is a list of expressions whose irst element is evaluated and if the result is not #f, the remaining expressions are evaluated and the value of the last one is returned as the value of the clause, otherwise the cond clause evaluates to #f.

*Usage:* (cond $\text{clause}_1$ ... $\text{clause}_n$)

*Example:* (cond ((= i 1) expr ...)

```
                 ((< i 1) expr ...)
                 ((> i 1) expr ...))
```

**cons**
*C Information:* SS_cons, SS_nargs
*Procedure:* Return a new pair whose car and cdr are the arguments.

*Usage:* (cons $\text{expr}_1$ $\text{expr}_2$)

*Example:* (cons 3 4) => (3 . 4)

```
          (cons 3 '(2 1)) => (3 2 1)
```

**cos**
*C Information:* SS_cos, SS_nargs
*Procedure:* Return the cosine of the argument.

**define**
*C Information:* SS_ident, SDEFINE, SS_sargs
*Macro:* Define variables and procedures in the current environment.

**define-macro**
*C Information:* SS_ident, SDEFINE, SS_sargs
*Macro:* Define a macro in the current environment.

**describe**
*C Information:* SS_describe, SS_nargs
*Procedure:* Print the documentation for a procedure to the specified device.

**display**
*C Information:* SS_display, SS_nargs
*Procedure:* Print an object to the specified device in human readable form.

**eof-object?**
*C Information:* SS_eofp, SS_sargs
*Procedure:* Return #t if the object is an end-of-file and #f otherwise.

**eq?**
*C Information:* SS_eq, SS_nargs
*Procedure:* Return #t iff the two objects are identical, i.e. the two objects are the same object.

*Usage:* (eq? $expr_1$ $expr_2$)

*Example:* (define foo 3)

```
(define bar 3)

(eq? foo bar) => #f

(eq? foo foo) => #t
```

**equal?**
*C Information:* SS_equal, SS_nargs
*Procedure:* Return the result of recursively applying eqv? to the arguments.

**eqv?**
*C Information:* SS_eqv, SS_nargs
*Procedure:* Return #t iff the two objects are equivalent, i.e. the values of the two objects are the same.

*Usage:* (eqv? $expr_1$ $expr_2$)

*Example:* (define foo 3)

```
(define bar 3)
```

```
(eqv? foo bar) => #t
```

**eval**
*C Information:* SS_eval, SS_sargs
*Procedure:* Evaluate the given form and return the value.

**even?**
*C Information:* SS_even, SS_nargs
*Procedure:* Return #t iff the argument is a number divisible exactly by 2.

**exp**
*C Information:* SS_exp, SS_nargs
*Procedure:* Return the exponential of the argument.

**expt**
*C Information:* SS_expt, SS_nargs
*Procedure:* Return the first argument raised to the power of the second.

**file?**
*C Information:* SS_filep, SS_sargs
*Procedure:* Return #t if the object is a file and #f otherwise.

**floor**
*C Information:* SS_floor, SS_nargs
*Procedure:* Return the greatest integer less than the argument.

**for-each**
*C Information:* SS_foreach, SS_nargs
*Procedure:* Apply a procedure to a set of lists (for side effect).

**guest-object?**
*C Information:* SS_guestp, SS_sargs
*Procedure:* Return #t if the object is a **GUEST** object and #f otherwise.

**if**
*C Information:* SS_ident, SIF, SS_sargs
*Macro:* Evaluate and return the consequent expression if the test evaluates to #t and evaluate and return the alternate expression otherwise.

*Usage:* (if test consequent alternate).

**input-port?**
*C Information:* SS_iportp, SS_sargs
*Procedure:* Return #t if the object is an input port (**IN_PORT**) and #f otherwise.

**integer?**
*C Information:* SS_intp, SS_sargs
*Procedure:* Return #t if the object is an integer number and #f otherwise.

**lambda**
*C Information:* SS_lambda, UR_MACRO, SS_nargs
*Macro:* Define an anonymous procedure.

*Usage:* (lambda parameters expression$_1$ ... expression$_n$).

**last**
*C Information:* SS_last, SS_sargs
*Procedure:* Return the last element of a list or return any other object.

**length**
*C Information:* SS_length, SS_nargs
*Procedure:* Return the number of elements in the given list.

**let**
*C Information:* SS_let, UE_MACRO, SS_nargs
*Macro:* Define a region with variables of local scope.

*Usage:* (let local-variables body)

*Example:* (let ((x (car '(3 . 4)))

```
      (y (cdr '(3 . 4))))
    (+ x y)) => 7
```

**let***
*C Information:* SS_letstr, UE_MACRO, SS_nargs
*Macro:* Define a region with variables of local scope. The variables can be used in the definition of other variables as soon as they are defined.

*Usage:* (let* local-variables body)

*Example:* (let* ((x (cdr ‘(1 2 3)))

```
(y (cdr x)))
(append x y)) => (2 3 3)
```

**list**
*C Information:* SS_list, SS_nargs
*Procedure:* Return a new list made up of the arguments.

**ln**
*C Information:* SS_ln, SS_nargs
*Procedure:* Return the natural logarithm of the argument.

**load**
*C Information:* SS_load, SS_sargs
*Procedure:* Open a file of Scheme forms and eval all the objects in it.

**log**
*C Information:* SS_log, SS_nargs
*Procedure:* Return the logarithm base 10 of the argument.

**make-new-symbol**
*C Information:* SS_newsym, SS_sargs
*Procedure:* Generate a new symbol.

**map**
*C Information:* SS_map, SS_nargs
*Procedure:* Map a procedure over a set of lists.

**member**
*C Information:* SS_member, SS_nargs
*Procedure:* Return the first sublist of the second arg whose car is equal? to the first arg.

**memq**
*C Information:* SS_memq, SS_nargs
*Procedure:* Return the first sublist of the second arg whose car is eq? to the first arg.

**memv**
*C Information:* SS_memv, SS_nargs
*Procedure:* Return the first sublist of the second arg whose car is eqv? to the first arg.

**negative?**
*C Information:* SS_neg, SS_nargs
*Procedure:* Return #t iff the argument is a number less than 0.

**newline**
*C Information:* SS_newline, SS_nargs
*Procedure:* Print a <CR><LF> or equivalent to the specified device.

**not**
*C Information:* SS_not, SS_sargs
*Procedure:* Return #t if object is #f and #f for any other object.

**null?**
*C Information:* SS_nullp, SS_sargs
*Procedure:* Return #t iff the object is the empty list, ().

**number?**
*C Information:* SS_numberp, SS_sargs
*Procedure:* Return #t if the object is a number and #f otherwise.

**odd?**
*C Information:* SS_odd, SS_nargs
*Procedure:* Return #t iff the argument is a number that is not even.

**open-input-file**
*C Information:* SS_opn_in, SS_sargs
*Procedure:* Open the specified file for input and return an **IN_PORT** object.

**open-output-file**
*C Information:* SS_opn_out, SS_sargs
*Procedure:* Open the specified file for output and return an **OUT_PORT** object.

**or**
*C Information:* SS_ident, SOR, SS_sargs
*Macro:* Evaluate the forms until one evaluates to something other than #f and return that

value as the result and return #f otherwise.

*Usage:* (or expression$_1$ ... expression$_n$)

*Example:* (or (= 3 4) (display "foo") (< 1 5) (display "bar")) => #t

**output-port?**
*C Information:* SS_oportp, SS_sargs
*Procedure:* Return #t if the object is an output port (**OUT_PORT**) and #f otherwise.

**pair?**
*C Information:* SS_pair, SS_args
*Procedure:* Return #t if the object is a cons or list and #f otherwise.

**positive?**
*C Information:* SS_pos, SS_nargs
*Procedure:* Return #t iff the argument is a number greater than 0.

**print-toggle**
*C Information:* SS_print_toggle, SS_zargs
*Procedure:* Toggle the printing of values.

**printf**
*C Information:* SS_printf, SS_nargs
*Procedure:* C-like formatted print function.

*Usage:* (printf port format . rest)

*Example:* (printf nil "%s - %d\n" "foo" 3) => #f

**procedure?**
*C Information:* SS_procp, SS_sargs
*Procedure:* Return #t if the object is a procedure object and #f otherwise.

**quasiquote**
*C Information:* SS_quasiq, UR_MACRO, SS_sargs
*Macro:* Like **quote** except that **unquote** and **unquote-splicing** forms are eval'd.

**quit**
*C Information:* SS_quit. UR_MACRO, SS_zargs

*Macro:* Exit from Scheme.

**quote**
*C Information:* SS_quote, UR_MACRO, SS_sargs
*Macro:* Return the expression without first evaluating it.

*Usage:* (quote expression)

**quotient**
*C Information:* SS_quotnt, SS_nargs
*Procedure:* Return quotient of two integers.

**read**
*C Information:* SS_read, SS_nargs
*Procedure:* Read an ASCII representation of an object and return the object.

**read-guest**
*C Information:* SS_rd_guest, SS_nargs
*Procedure:* Return a guest object using the supplied function rd_guest.

**real?**
*C Information:* SS_realp, SS_sargs
*Procedure:* Return #t if the object is a real number and #f otherwise.

**remainder**
*C Information:* SS_remnd, SS_nargs
*Procedure:* Return remainder of division of the two arguments.

**reset**
*C Information:* SS_reset, SS_zargs
*Procedure:* Unwind the Error/Break Stack and return to top level.

**return-level**
*C Information:* SS_retlev, SS_nargs
*Procedure:* Pop n levels off the Error/Break Stack and return the second arg as value.

**reverse**
*C Information:* SS_reverse, SS_sargs
*Procedure:* Destructively reverse the list and return it.

**set!**
*C Information:* SS_ident, SSET, SS_sargs
*Macro:* Replace the contents of the location the variable points to with the value.

**set-car!**
*C Information:* SS_setcar, SS_nargs
*Procedure:* Replace the car of the first argument with the second and return the new car.

**set-cdr!**
*C Information:* SS_setcdr, SS_nargs
*Procedure:* Replace the cdr of the first argument with the second and return the new cdr.

**sin**
*C Information:* SS_sin, SS_nargs
*Procedure:* Return the sine of the argument.

**sqrt**
*C Information:* SS_sqrt, SS_nargs
*Procedure:* Return the principal square root of the argument.

**stats-toggle**
*C Information:* SS_stats_toggle, SS_zargs
*Procedure:* Toggle the printing of control statistics.

**string?**
*C Information:* SS_strp, SS_sargs
*Procedure:* Return #t if the object is a string and #f otherwise.

**symbol?**
*C Information:* SS_varp, SS_sargs
*Procedure:* Return #t if the object is a variable and #f otherwise.

**system**
*C Information:* SS_system, SS_sargs
*Procedure:* Have the operating system carry out the command contained in the string.

**tan**

*C Information:* SS_tan, SS_nargs
*Procedure:* Return the tangent of the argument.

**trace**
*C Information:* SS_trace, SS_nargs
*Procedure:* Trace calls to the procedures in the list of arguments.

**transcript-off**
*C Information:* SS_trans_off, UR_MACRO, SS_zargs
*Macro:* Close the transcript file to stop recording a Scheme session.

**transcript-on**
*C Information:* SS_trans_on, SS_sargs
*Procedure:* Open the specified transcript file to start recording a Scheme session.

**truncate**
*C Information:* SS_trunc, SS_nargs
*Procedure:* Return the integer resulting from the truncation of the argument.

**unquote**
*C Information:* SS_unquote, UR_MACRO, SS_sargs
*Macro:* In a quasiquote'd form inserts the result of evaluating its argument.

**unquote-splicing**
*C Information:* SS_unq_spl, UR_MACRO, SS_sargs
*Macro:* In a quasiquote'd form splices the evaluated list into the quoted form.

**untrace**
*C Information:* SS_untrace, SS_nargs
*Procedure:* Stop tracing the procedures in the argument list.

**write**
*C Information:* SS_write, SS_nargs
*Procedure:* Put the printed representation of an object to the specified device.

**zero?**
*C Information:* SS_zero, SS_nargs
*Procedure:* Return #t iff the argument is a number equal to 0.

## The Large Set

**call-with-input-file**
*C Information:* SS_call_if, SS_nargs
*Procedure:* Open the named file and evaluate a procedure using the port for input. This is analogous to input redirection in certain operating systems

*Usage:* (call-with-input-file string procedure)

*Example:* (call-with-input-file "foobar.scm" my-reader)

**call-with-output-file**
*C Information:* SS_call_of, SS_nargs
*Procedure:* Open the named file and evaluate a procedure using the port for output. This is analogous to output redirection in certain operating systems.

*Usage:* (call-with-output-file string procedure)

**char?**
*C Information:* SS_charp, SS_sargs
*Procedure:* Return #t iff the object is of type char.

**char=?**
*C Information:* SS_chreq, SS_nargs
*Procedure:* Return #t iff the chars are the same.

**char>=?**
*C Information:* SS_chrge, SS_nargs
*Procedure:* Return #t iff the first character is ‘greater than or equal to’ the second.

**char>?**
*C Information:* SS_chrgt, SS_nargs
*Procedure:* Return #t iff the first character is ‘greater than’ the second.

**char<=?**
*C Information:* SS_chrle, SS_nargs
*Procedure:* Return #t iff the first character is ‘less than or equal to’ the second.

**char<?**

*C Information:* SS_chrlt, SS_nargs
*Procedure:* Return #t iff the first character is ‘less than’ the second.

**char->integer**
*C Information:* SS_chrint, SS_sargs
*Procedure:* Return the integer representation of the given integer.

**current-input-port**
*C Information:* SS_curr_ip, SS_zargs
*Procedure:* Return the current default input port.

**current-output-port**
*C Information:* SS_curr_op, SS_zargs
*Procedure:* Return the current default output port.

**define-global**
*C Information:* SS_define_global, UE_MACRO, SS_nargs
*Macro:* Define variables and procedures in the global environment.

**hash-dump**
*C Information:* SS_hash_dump, SS_sargs
*Procedure:* Return a list of the names in the given hash table.

**hash-element?**
*C Information:* SS_hashelp, SS_sargs
*Procedure:* Return #t if the object is a hash-element.

**hash-info**
*C Information:* SS_hash_info, SS_sargs
*Procedure:* Return the specified hash table’s size, number of hash-elements, and documentation flag.

**hash-install**
*C Information:* SS_hash_install, SS_nargs
*Procedure:* Install the given object in the given hash table.

**hash-lookup**
*C Information:* SS_hash_lookup, SS_nargs
*Procedure:* Look up and return the named object in the given hash table.

**hash-remove**
*C Information:* SS_hash_remove, SS_nargs
*Procedure:* Remove the named object from the given hash table.

**hash-table?**
*C Information:* SS_hashtabp, SS_sargs
*Procedure:* Return #t if the object is a hash table.

**integer->char**
*C Information:* SS_intchr, SS_sargs
*Procedure:* Return the character representation of the given integer.

**list->string**
*C Information:* SS_lststr, SS_nargs
*Procedure:* Return a string constructed from a list of characters.

**list->vector**
*C Information:* SS_lstvct, SS_sargs
*Procedure:* Return a vector whose elements are the same as the list's.

**make-hash-table**
*C Information:* SS_make_hash_table, SS_sargs
*Procedure:* Return a new hash table.

**make-vector**
*C Information:* SS_mkvect, SS_sargs
*Procedure:* Return a new vector whose length is specified by the argument.

**process?**
*C Information:* SS_prp, SS_sargs
*Procedure:* Return #t if the object is a **PROCESS_OBJ**.

**process-close**
*C Information:* SS_cls_pr, SS_sargs
*Procedure:* Terminate a process.

**process-open**

*C Information:* SS_opn_pr, SS_nargs
*Procedure:* Exec and return a process object which can be used much like a port in those I/O functions which take a process object as the source or sink of messages.

*Usage:* (process-open string)

*Example:* (define pp (process-open "ls"))

**process-read-line**
*C Information:* SS_pr_rd_line, SS_sargs
*Procedure:* Return a string received from a process.

*Usage:* (process-read-line process)

*Example:* (print (process-read-line pp))

**process-running?**
*C Information:* SS_pr_runp, SS_sargs
*Procedure:* Return #t if the process is still running.

**process-send-line**
*C Information:* SS_pr_sn_line, SS_nargs
*Procedure:* Send a string to a process.

**process-status**
*C Information:* SS_pr_stat, SS_sargs
*Procedure:* Return a list of the process id, in, out, status, and reason members of the **PROCESS** structure.

**read-char**
*C Information:* SS_rd_chr, SS_nargs
*Procedure:* Read and return a single character.

**read-line**
*C Information:* SS_rd_line, SS_nargs
*Procedure:* Read a line of text and return a string.

**string=?**
*C Information:* SS_streq, SS_nargs
*Procedure:* Return #t iff the strings are the same (length too).

**string>=?**
*C Information:* SS_strge, SS_nargs
*Procedure:* Return #t iff the first string is ‘greater than or equal to’ the second.

**string>?**
*C Information:* SS_strgt, SS_nargs
*Procedure:* Return #t iff the first string is ‘greater than’ the second.

**string<=?**
*C Information:* SS_strle, SS_nargs
*Procedure:* Return #t iff the first string is ‘less than or equal to’ the second.

**string<?**
*C Information:* SS_strlt, SS_nargs
*Procedure:* Return #t iff the first string is ‘less than’ the second.

**string->list**
*C Information:* SS_strlst, SS_sargs
*Procedure:* Construct a list of the characters in the given string.

**string->port**
*C Information:* SS_strprt, SS_sargs
*Procedure:* Encapsulate a string as a pseudo input port for reading.

**string->symbol**
*C Information:* SS_strsym, SS_sargs
*Procedure:* Make a new variable with name given by the string.

**string-append**
*C Information:* SS_strapp, SS_nargs
*Procedure:* Append the argument strings together into a new string and return it.

**string-length**
*C Information:* SS_strlen, SS_sargs
*Procedure:* Return the number of characters in the given string.

**string-ref**
*C Information:* SS_strref, SS_nargs

*Procedure:* Return the nth character in the given string.

**substring**
*C Information:* SS_strsub, SS_nargs
*Procedure:* Extract the substring zero-origin indexed by the last two args.

**symbol->string**
*C Information:* SS_symstr, SS_sargs
*Procedure:* Make a new string out of the given variable name.

**vector**
*C Information:* SS_vector, SS_nargs
*Procedure:* Analog to list procedure for vectors.

**vector?**
*C Information:* SS_vectp, SS_sargs
*Procedure:* Return #t iff the object is of type vector.

**vector-length**
*C Information:* SS_vctlen, SS_sargs
*Procedure:* Return the number of elements in the specified vector.

**vector->list**
*C Information:* SS_vctlst, SS_sargs
*Procedure:* Return a list whose elements are the same as the vector's.

**vector-ref**
*C Information:* SS_vctref, SS_nargs
*Procedure:* Return the nth element of the given vector.

**vector-set!**
*C Information:* SS_vctset, SS_nargs
*Procedure:* Sets the nth element of the given vector.

**write-char**
*C Information:* SS_wr_chr, SS_nargs
*Procedure:* Write a single character to the specified port.

# The PACT Scheme Library

In this section we list some of the functions in the PACT SCHEME library. Although there are many functions here, only a relative handful are useful to application developers wishing to integrate the interpreter into their product. All of the relevant declarations are made in the header file scheme.h which should be #include'd in your source code.

## Constants

These #define'd constants are part of the essential glue of PACT SCHEME and the other parts of PACT which use it (i.e. SX and ULTRA).

### Object Type Constants

These constants define the type indeces for SCHEME objects. They are used in conjunction with **SS_args, SS_call_scheme**, and **SS_make_list**. They are also implicitly used by the C and SCHEME level predicate functions.

| | | |
|---|---|---|
| SS_OBJECT_I | 10 | Generic object type |
| CONS | 11 | Cons object type |
| VARIABLE | 12 | Symbol object type |
| PROC_OBJ | 13 | Procedure object type |
| BOOLEAN | 14 | Boolean object type |
| IN_PORT | 15 | Input port object type |
| OUT_PORT | 16 | Output port object type |
| VECTOR | 19 | Vector object type |
| CHAR_OBJ | 20 | Character object type |
| HASH_ELEMENT | 21 | Hash element object type |
| HASH_TABLE | 22 | Hash table object type |
| PROCESS_OBJ | 23 | Process object type |

### Evaluation Type Constants

The following #define'd constants are described in detail in the section on interfacing compiled and interpreted code.

| | | |
|---|---|---|
| SELF_EV | 1 | Objects evaluate to themselves |
| VAR_EV | 2 | Symbol objects evaluate to their value binding |
| PROC_EV | 3 | Procedure call (conses are the only things that have this) |

### Procedure Type Constants

These constants are used to distinguish among different ways of handling procedures and their calls. They are described in more detail in the section on interfacing compiled and interpreted code.

| | | |
|---|---|---|
| SS_PR_PROC | 'z' | Evaluate arguments, don't re-evaluate the result |
| SS_EE_MACRO | 'p' | Evaluate arguments, re-evaluate the result |
| SS_UR_MACRO | 'r' | Don't evaluate the arguments, don't re-evaluate the result |
| SS_UE_MACRO | 'q' | Don't evaluate the arguments, re-evaluate the result |

## Macros

These macros facilitate interfacing with the interpreter and writing C level routines to be used by the interpreter.

**Object Accessors**

| | |
|---|---|
| SS_GET(type, x) | cast the value part of x to be a type pointer and return it |
| SSS_INQUIRE_OBJECT(name) | lookup name in the system table and return a pointer to the associated object |
| SS_OBJECT_GC(x) | return the reference count of x (how many things point to it) |
| SS_UNCOLLECT(x) | make sure that x can never be garbage collected |
| SS_OBJECT_TYPE(x) | return the type index of x |
| SS_OBJECT_NAME(x) | return the print name of x |
| SS_OBJECT(x) | return the value part of x |
| SS_OBJECT_ETYPE(x) | return the evaluation type of x |
| SS_OBJECT_PRINT(x, strm) | invoke the print method of x on the output port strm |
| SS_OBJECT_FREE(x) | invoke the release method of x |

**Number Accessors**

| | |
|---|---|
| SS_INTEGER_VALUE(x) | return the value of the integer object x (as a long) |
| SS_FLOAT_VALUE(x) | return the value of the float object x (as a double) |
| SS_int_val(x) | return TRUE iff the number x has an integer value |

**Input_port/output_port Accessors**

| | |
|---|---|
| SS_OUTSTREAM(x) | return the FILE * from the output port x |
| SS_INSTREAM(x) | return the FILE * from the input port x |

**Cons Accessors**

| | |
|---|---|
| SS_CAR_MACRO(x) | return the car of x (no error checking) |
| SS_CDR_MACRO(x) | return the cdr of x (no error checking) |

**Variable Accessors**

| | |
|---|---|
| SS_VARIABLE_NAME(x) | return the name of the symbol object x |
| SS_VARIABLE_VALUE(x) | return the value of the symbol object x |

**String Accessors**

| | |
|---|---|
| SS_STRING_TEXT(x) | return the text of the string object x |
| SS_STRING_LENGTH(x) | return the length of the text of x |

**Procedure Accessors**

| | |
|---|---|
| SS_PROCEDURE_TYPE(x) | return the type of the procedure object x |
| SS_PROCEDURE_NAME(x) | return the name of the procedure object x |
| SS_PROCEDURE_DOC(x) | return the documentation for the procedure x |
| SS_PROCEDURE_TRACEDP(x) | return TRUE if the procedure x is traced |
| SS_PROCEDURE_PROC(x) | return the actual procedure for x |

**Vector Accessors**

| | |
|---|---|
| SS_VECTOR_LENGTH(x) | return the length of the vector object x |
| SS_VECTOR_ARRAY(x) | return the array of objects of the vector x |

**Character Accessor**

| | |
|---|---|
| SS_CHARACTER_VALUE(x) | return the value of the character object x |

**Process Accessors**

| | |
|---|---|
| SS_PROCESS_VALUE(x) | return the PROCESS * of the process object x |
| SS_PROCESS_STATUS(x) | return the status of the process x |

**Predicates**

| | |
|---|---|
| SS_integerp(x) | TRUE iff x is of type INTEGER |
| SS_floatp(x) | TRUE iff x is of type FLOAT |
| SS_stringp(x) | TRUE iff x is of type STRING |

| | |
|---|---|
| SS_consp(x) | TRUE iff x is of type CONS |
| SS_variablep(x) | TRUE iff x is of type VARIABLE (symbol) |
| SS_inportp(x) | TRUE iff x is of type IN_PORT (input port) |
| SS_outportp(x) | TRUE iff x is of type OUT_PORT (output port) |
| SS_eofobjp(x) | TRUE iff x is of type EOF_OBJ (#eof) |
| SS_nullobjp(x) | TRUE iff x is of type NULL_OBJ (() or nil) |
| SS_procedurep(x) | TRUE iff x is of type PROC_OBJ |
| SS_booleanp(x) | TRUE iff x is of type BOOLEAN (#t or #f) |

The following are available with the LARGE option (default):

| | |
|---|---|
| SS_hash_tablep(x) | TRUE iff x is of type HASH_TABLE |
| SS_hash_elementp(x) | TRUE iff x is of type HASH_ELEMENT |
| SS_charobjp(x) | TRUE iff x is of type CHAR_OBJ |
| SS_vectorp(x) | TRUE iff x is of type VECTOR |
| SS_processp(x) | TRUE iff x is of type PROCESS_OBJ |

**Memory Management**

| | |
|---|---|
| SS_GC(x) | decrement the reference count of x and release it if the count is zero |
| SS_mark(x) | increment the reference count of x |
| SS_Assign(x, val) | assign object x to object val making sure to get the reference counts right |

**Execution Control**

| | |
|---|---|
| SS_set_cont(to_go, to_return) | setup a new continuation |
| SS_go_cont | go to the current continuation |
| SS_jump(x) | jump to the label x |
| SS_tracedp(x) | return the trace field in the procedure struct x |

**Stack Control**

| | |
|---|---|
| SS_Save(x) | save x on the stack |
| SS_Restore(x) | pop an object off the stack into x and release the stack entry |

## Variables

**Strings**

| | |
|---|---|
| SS_OBJECT_S | string containing "object" |
| SS_POBJECT_S | string containing "object *" |
| SS_prompt | the prompt which is displayed by the interpreter |
| SS_ans_prompt | the preamble to the interpreter's printing of the result |

**Objects**

| | |
|---|---|
| SS_quoteproc | pointer to the quote procedure |
| SS_indev | pointer to the standard input port object (stdin equivalent) |
| SS_outdev | pointer to the standard output port object (stdout equivalent) |
| SS_histdev | pointer to the history output port object (see transcript) |
| SS_null | pointer to the () object |
| SS_eof | pointer to the #eof object |
| SS_t | pointer to the #t object |
| SS_f | pointer to the #f object |
| SS_else | pointer to the else object |

## Handlers

The following functions are described in detail in the section on interfacing compiled and interpreted code.

**Function Handlers**

**object *SS_zargs(object *argl)**

**object *SS_sargs(object *argl)**

**object *SS_nargs(object *argl)**

**Variable Handlers**

It is possible to give the following SCHEME level functional interpretation to compiled code variables: suppose you have a variable x bound to the value 3 then:

(x) => 3

(x 2) => 2

and x is now bound to 2. These handlers are supplied so that you may install your application variables as implicit functions. They are used with SS_install_cf.

**object *SS_acc_REAL(byte *vr, object *argl)**

**object *SS_acc_int(byte *vr, object *argl)**

**object *SS_acc_long(byte *vr, object *argl)**

**object *SS_acc_char(byte *vr, object *argl)**

**object *SS_acc_string(byte *vr, object *argl)**

**object *SS_acc_ptr(byte *vr, object *argl)**

**object *SS_acc_var(byte *vr, object *argl, int otype)**

## Functions

The following functions are just some of the many functions in the PACT SCHEME library. They are selected because they are the ones that developers will use to integrate the interpreter into their applications. The remainder of the functions declared in scheme.h are C level implementations of SCHEME level functions and as such are only accessed by the interpreter.

### Memory Management

These functions support the creation and destruction of SCHEME objects. Although these handle the native PACT SCHEME data types, they also provide some guidance for developers who wish to make their own data types known to the interpreter.

**object *SS_mk_variable(char *name, object *v)**

**object *SS_mk_string(char *s)**

**object *SS_mk_inport(FILE *str)**

**object *SS_mk_outport(FILE *str)**

**object *SS_mk_integer(long i)**

**object *SS_mk_float(double d)**

**object *SS_mk_cons(object *car, object *cdr)**

**object *SS_mk_char(int i)**

**object *SS_mk_vector(int l)**

**object *SS_mk_hash_table(HASHTAB *tb)**

**object *SS_mk_hash_element(hashel *hp)**

**object *SS_mk_object(byte *p, int type, int ev_type, char *name)**

**SS_mk_object** creates and returns a new object of type type which has the pointer p as its value. The object type is taken from the list of defined type indeces some of which are discussed in the section Object Type Constants above. Also specified are its evaluation type (ev_type) which is required and its print name (name) if any. The evaluation type is one of the values from the Evaluation Type Constants section above. All of the SS_mk_xxxx functions above call **SS_mk_object** as the final step to wrap an object around whatever other data type they create and initialize.

**void SS_rl_object(object *obj)**

**List Processors**

These functions support applications in dealing with ubiquitous lists at the C level.

**object *_SS_setcar(object *pair, object *car)**

**object *_SS_setcdr(object *pair, object *cdr)**

These functions set the value of the car or cdr of the specified cons object.

**object *SS_car(object *obj)**

**object *SS_cdr(object *obj)**

**object *SS_caar(object *obj)**

**object *SS_cadr(object *obj)**

**object *SS_cdar(object *obj)**

**object *SS_cddr(object *obj)**

**object *SS_caaar(object *obj)**

**object *SS_caadr(object *obj)**

**object *SS_cadar(object *obj)**

**object *SS_caddr(object *obj)**

**object *SS_cdaar(object *obj)**

**object *SS_cdadr(object *obj)**

**object *SS_cddar(object *obj)**

**object *SS_cdddr(object *obj)**

These functions access parts of lists including imbedded lists.

**object *_SS_lst_tail(object *lst, int n)**

This function is a more general list accessor than the above, since it lets you go to arbitrary depth in the list. It returns the list starting at the nth element of the given list. Used in conjunction with SS_car, all lists can be picked apart this way.

**int _SS_length(object *lst)**

Return the length of the given list.

**Input/Output**

These functions handle most of the I/O chores which straddle the interpreter and the application.

**void SS_unget_ch(int c, object *str)**

**void SS_put_ch(int c, object *str)**

**int SS_get_ch(object *str, int ign_ws)**

These functions are used analogously to getc, putc, and ungetc in the C standard library. They are provided as the standard operators used by the SCHEME reader. Applications may define their own and attach them to the relevant hooks instead of these.

**void SS_wr_atm(object *obj, object *strm)**

Print an atomic object to the specified output port object.

**object *_SS_print( object *obj,<br>char *begin, char *end,<br>object *port)**

Print the given object to the given port preceded by begin and followed by end.

**object *SS_load(object *port)**

Read and evaluate the expressions in the specified input port (usually a file).

**C/SCHEME Interface**

These are the functions used to communicate between the interpreter and compiled code. See the section on interfacing interpreted and compiled code later in this manual.

**object *SS_make_list(int first, ...)**

Given a list of type index, variable pointer pairs create and return a SCHEME list. A pair with a type index of 0 terminates the list. The pointers in the pairs must be pointers to the types indicated by the type index. Thus SC_INTEGER_I would be accompanied by and int *.

**object *SS_call_scheme(char *func, ...)**

Call the function named by func with the arguments supplied. The arguments are specified the same way that they are for **SS_make_list**.

**int SS_args(object *s, ...)**

Extract arguments from the specified object which may be atomic or a list. The objects to be extracted are specified as for **SS_make_list** and **SS_call_scheme** as type index, pointer pairs. A type index of zero terminates the list. The number of items successfully extracted is returned. Only one thing can be extracted from an atomic object. It is not required that the length of the list match the number of specifications in the call. The extraction terminates at the earlier of the exhaustion of the list s or the end of the specifications for extraction.

**int SS_run(char *s)**

Parse and evaluate a single expression represented by s.

**int SS_load_scm(char *name)**

Open and load (in the sense of **SS_load**) the file specified by name.

**Top Level**

These functions are high level routines for initializing the interpreter and entering into read-eval-print loops.

**void SS_inst_prm(byte)**

**void SS_repl(byte)**

**void SS_end_scheme(int val)**

**void SS_init_scheme(char *Code, char *Vers)**

**void SS_init_path(byte)**

**void SS_inst_const(byte)**

**void SS_init_stack(byte)**

**void SS_init_cont(byte)**

**void SS_interrupt_handler(int sig)**

**Error Handling**

These routines are for controlling error handling.

**int SS_err_catch(PFInt func, PFInt errf)**

**object *SS_pop_err(int n, int flag)**

**void SS_push_err(int flag, int type)**

**void SS_error(char *s, object *obj)**

**void _SS_restore_state(object *esc_proc)**

**void _SS_print_err_msg(FILE *str, char *s, object *obj)**

**void _SS_print_err_msg_a(FILE *str, char *s, object *obj)**

**Helpers**

**int _SS_numberp(object *obj)**

**char *SS_get_string(object *obj)**

**Registration Routines**

These routines make C level objects known to the interpreter.

**void SS_install( char *name, char *doc,**

**PFPObject hand, PFPObject proc,**
**int type)**

Install a procedure in the system table under the name, name, and with documentation, doc. The handler and procedure (hand and proc respectively) must be previously declared. The procedure type is one of those given in the Procedure Type Constants section above.

**object *SS_install_cf( char *name, char *doc,**
**PFPObject handler, ...)**

Install a variable in the system table under the name, name, and with documentation, doc. The handler, hand, must be one of those given in the Variable Handlers section above.

**object *SS_install_cv(char *name, byte *pval, int type)**

Install a variable in the system table to be used as a conventional SCHEME variable. This is equivalent to doing (define name value) at the SCHEME level. It simply lets you compile into the application any variables that you wish to handle at the compiled and interpreted level.

# Syntax Modes in SCHEME

The PACT SCHEME interpreter has the hooks to allow applications to install alternate parsers. The effect of this is to allow the interpreter to interpret other languages. The idea is to write a YACC-style grammar and perhaps a LEX-style analyzer to parse an input language. The grammar actions in effect translate the input language into SCHEME for evaluation.

As stated before this can be used to create an interpreter for your favorite language. Alternatively it can be thought of as a means of translating your (second) favorite language into SCHEME.

To add a mode, you supply the appropriate parser and analyzer. You supply a function to switch the reader into the new syntax mode. You can register a file suffix to tell the interpreter to switch to your new mode when loading source files with that suffix.

As an example of how this works a C syntax mode is supplied. Other modes can be patterned after C mode.

## C Syntax Mode

C mode is based on the grammar of ANSI standard C. It features all of the syntax of ANSI standard C except:

**enum**

type **char** ; although **char *** is supported

multi-dimensional arrays semantics (syntax is there)

**a.b** syntax ; although **a->b** is supported

**&** and ***** ; i.e. address of and dereference

Also the C preprocessor directives are not included at this time.

Because C identifiers are more restricted that SCHEME identifiers, the following conventions are used to map names of SCHEME functions into a form which the C parser/analyzer will accept:

| **SCHEME** | **C** |
|---|---|
| - | _ |
| * | _ |
| ? | p |
| ! | b |
| -> | _to_ |

Files ending in **.c** or **.h** are interpreted in C mode.

Because SCHEME objects carry their own type information, the use of types in C mode should be thought of more as a reminder to you than as a mandate to the interpreter. For example:

```
void foo(char *name)
   {void *x;
    int y;

    x = open_input_file(name);
    if (input_portp(x))
       close_input_file(x);

    return;}
```

and

```
void foo(int name)
   {int x;
    int y;

    x = open_input_file(name);
    if (input_portp(x))
       close_input_file(x);

    return;}
```

in spite of having wildly different types for the variables *name* and *x* behave the same as the SCHEME coding

```
(define (foo name)
   (let* (x y)
         (set! x (open-input-file name))
         (if (input-port? x)
             (close-input-file x))))
```

The first version should be preferred because the coding more clearly states in C terms the intentions for the data types. That is, **char *name** is a better way (in C at least) to think of a character array than **int name**.

Notice also the function names in compliance with the convention stated above.

It is also possible to mix SCHEME and C syntax in a file. There are two ways to do this. First by calling a function to set the parsing mode. For example:

```
(scheme-mode)

(define (foo x)
    (+ x 3))

(c-mode)

foo(2);

scheme_mode();

(foo 2)

(c-mode)

quit();
```

The second method of mixing SCHEME and C is special to the C mode. The SCHEME interpreter has been programmed to look for **{** and switch to C mode until the corresponding **}**. For example:

```
(define x 3)

(if (= x 3)
    {int y;
     y = x + 2;
     printf(nil, "%s\n", y);})
```

This is handy especially for doing complicated arithmetic expressions in algebraic (infix) notation instead of Polish (prefix) notation.

# SCHEME + EXTENSIONS

The functions of SX are presented in groupings which correspond to the PACT functionalities to which they interface. The reader is assumed to be familiar with the functionality and structures of the part of PACT which these functions cover. See the end of this manual for references to the other PACT documentation.

## SX Procedures

### Pure SX Functions

**memory-trace**
*C Information:* SX_mem_trace, SS_zargs
*Procedure:* Return the number of allocated memory blocks.

**pp**
*C Information:* SX_pp_names, SS_sargs
*Procedure:* Print a list in nice even columns.

### PML Functions

**list->pm-array**
*C Information:* SX_list_array, SS_nargs
*Procedure:* Return a numeric array built from a list of numbers.

**pm-array?**
*C Information:* SX_numeric_arrayp, SS_sargs
*Procedure:* Return #t if the object is a numeric array and #f otherwise.

**pm-array->list**
*C Information:* SX_array_list, SS_sargs
*Procedure:* Return a list of numbers built from a numeric array.

**pm-array-length**
*C Information:* SX_num_arr_len, SS_sargs
*Procedure:* Return the length of the given numeric array.

**pdbdata->pm-mapping**
*C Information:* SX_pdbdata_mapping, SS_nargs
*Procedure:* Read a PML mapping object from a PDB file.

> *Usage:* (pdbdata->pm-mapping file name)

**pm-grotrian-mapping?**
*C Information:* SX_grotrian_mappingp, SS_sargs
*Procedure:* Return #t if the object is a PML grotrian mapping, and #f otherwise.

**pm-make-mapping**
*C Information:* SX_make_pml_mapping, SS_nargs
*Procedure:* Return a PML mapping.

*Usage:* (pm-make-mapping domain range [centering category name emap next])

**pm-make-set**
*C Information:* SX_make_pml_set, SS_nargs
*Procedure:* Return a PML set.

*Usage:* (pm-make-set name mesh-shape-list element-arrays)

**pm-mapping?**
*C Information:* SX_mappingp, SS_sargs
*Procedure:* Return #t if the object is a PML mapping, and #f otherwise.

**pm-mapping->pdbdata**
*C Information:* SX_mapping_pdbdata, SS_nargs
*Procedure:* Write a PML mapping object to a PDB file.

*Usage:* (pm-mapping->pdbdata mapping file)

**pm-mapping-dimension**
*C Information:* SX_get_dimension, SS_sargs
*Procedure:* Return a pair containing the dimensionality of the domain and range of the specified mapping.

**pm-mapping-domain**
*C Information:* SX_get_domain, SS_sargs
*Procedure:* Return the domain of the given mapping.

**pm-mapping-name**
*C Information:* SX_get_mapping_name, SS_sargs
*Procedure:* Return the name of the given mapping.

**pm-mapping-range**

*C Information:* SX_get_range, SS_sargs
*Procedure:* Return the range of the given mapping.

**pm-scale-domain**
*C Information:* SX_scale_domain, SS_nargs
*Procedure:* Multiply all components of a mapping domain by a scalar value.

**pm-scale-range**
*C Information:* SX_scale_range, SS_nargs
*Procedure:* Multiply all components of a mapping range by a scalar value.

**pm-set?**
*C Information:* SX_setp, SS_sargs
*Procedure:* Return #t if the object is a PML set, and #f otherwise.

**pm-set-mapping-type**
*C Information:* SX_set_map_type, SS_nargs
*Procedure:* Set the type of a mapping object to the given string.

**pm-set-volume**
*C Information:* SX_set_volume, SS_sargs
*Procedure:* Return the product of the extrema of the given set.

**pm-shift-domain**
*C Information:* SX_shift_domain, SS_nargs
*Procedure:* Add a scalar value to all components of a mapping domain.

**pm-shift-range**
*C Information:* SX_shift_range, SS_nargs
*Procedure:* Add a scalar value to all components of a mapping range.

**Math Functions on Mappings**

**+**
*C Information:* PM_fplus, _SX_mh_b_s,
*Procedure:* Sum of range values, i.e. a+b.

*Usage:* + a [b ...]

**-**
*C Information:* PM_fminus, _SX_mh_b_s,
*Procedure:* Difference of range values, i.e. a-b. If there is only one argument the result is unary negation.

> *Usage:* - a [b ...]

*****
*C Information:* PM_ftimes, _SX_mh_b_s,
*Procedure:* Product of range values, i.e. a*b. If there is only one argument this is an identity operation.

> *Usage:* * a [b ...]

**/**
*C Information:* PM_fdivide, _SX_mh_b_s,
*Procedure:* Quotient of range values, i.e. a/b. If there is only one argument this takes the multiplicative inverse of the mapping.

> *Usage:* / a [b ...]

**abs**
*C Information:* ABS, _SX_mh_u_s,
*Procedure:* Absolute value of range values.

> *Usage:* abs mappings

**acos**
*C Information:* acos, _SX_mh_u_s,
*Procedure:* Arccos of range values.

> *Usage:* acos mappings

**asin**
*C Information:* asin, _SX_mh_u_s,
*Procedure:* Arcsin of range values.

> *Usage:* asin mappings

**atan**
*C Information:* atan, _SX_mh_u_s,

*Procedure:* Arctan of range values.

*Usage:* atan mappings

**cos**
*C Information:* cos, _SX_mh_u_s,
*Procedure:* Cosine of range values.

*Usage:* cos mappings

**cosh**
*C Information:* cosh, _SX_mh_u_s,
*Procedure:* Hyperbolic cosine of range values.

*Usage:* cosh mappings

**exp**
*C Information:* exp, _SX_mh_u_s,
*Procedure:* Exponential of range values.

*Usage:* exp mappings

**integrate**
*C Information:* SX_integrate_mapping, _SX_mh_u_s_nm,
*Procedure:* Compute the integrals of the given mappings.

*Usage:* integrate mappings

**j0**
*C Information:* j0, _SX_mh_u_s,
*Procedure:* Zeroth order Bessel function of the first kind on range values.

*Usage:* j0 mappings

**j1**
*C Information:* j1, _SX_mh_u_s,
*Procedure:* First order Bessel function of the first kind on range values.

*Usage:* j1 mappings

**jn**

*C Information:* PM_jn, _SX_mh_u_s,
*Procedure:* Nth order Bessel function of the first kind on range values.

*Usage:* jn mappings n

**ln**
*C Information:* PM_ln, _SX_mh_u_s,
*Procedure:* Natural log of range values.

*Usage:* ln mappings

**log10**
*C Information:* log10, _SX_mh_u_s,
*Procedure:* Base 10 log of range values.

*Usage:* log10 mappings

**norm**
*C Information:* SX_norm_mapping, _SX_mh_u_s_nm,
*Procedure:* Compute the euclidean norms of the given mappings.

*Usage:* norm mappings

**random**
*C Information:* PM_random, _SX_mh_u_s,
*Procedure:* Generate random range values between -1 and 1 for the mappings.

*Usage:* random mappings

**recip**
*C Information:* PM_recip, _SX_mh_u_s,
*Procedure:* Reciprocal of range values.

*Usage:* recip mappings

**sin**
*C Information:* sin, _SX_mh_u_s,
*Procedure:* Sine of range values.

*Usage:* sin mappings

**sinh**
*C Information:* sinh, _SX_mh_u_s,
*Procedure:* Hyperbolic sine of range values.

*Usage:* sinh mappings

**sqr**
*C Information:* PM_sqr, _SX_mh_u_s,
*Procedure:* Square of range values.

*Usage:* sqr mappings

**sqrt**
*C Information:* PM_sqrt, _SX_mh_u_s,
*Procedure:* Square root of range values.

*Usage:* sqrt mappings

**tan**
*C Information:* tan, _SX_mh_u_s,
*Procedure:* Tangent of range values.

*Usage:* tan mappings

**tanh**
*C Information:* tanh, _SX_mh_u_s,
*Procedure:* Hyperbolic tangent of range values.

*Usage:* tanh mappings

**y0**
*C Information:* y0, _SX_mh_u_s,
*Procedure:* Zeroth order Bessel function of the second kind on range values.

*Usage:* y0 mappings

**y1**
*C Information:* y1, _SX_mh_u_s,
*Procedure:* First order Bessel function of the second kind on range values.

*Usage:* y1 mappings

**yn**
*C Information:* PM_yn, _SX_mh_u_s,
*Procedure:* Nth order Bessel function of the second kind on range values.

*Usage:* yn mappings n

## PDB Functions

The functions in this section are divided into two parts. The first part deals with PDB functionality somewhat in parallel to the PDB C API. The second part describes functions that go beyond the PDB C API and allow more arbitrary binary files to be accessed with lower levl PDB machinery.

### High Level API Functions

**change-directory**
*C Information:* SX_change_directory, SS_nargs
*Procedure:* Change the current PDB file directory.

*Usage:* (change-directory file directory)

**close-pdbfile**
*C Information:* SX_close_pdbfile, SS_sargs
*Procedure:* Close a PDB file.

*Usage:* (close-pdbfile file)

**create-link**
*C Information:* SX_create_link, SS_nargs
*Procedure:* Create a link to a variable in a PDB file.

*Usage:* (create-link file variable-name link-name)

**create-pdbfile**
*C Information:* SX_create_pdbfile, SS_sargs
*Procedure:* Create and return a PDB file.

*Usage:* (create-pdbfile expression)

**current-directory**
*C Information:* SX_current_directory, SS_sargs
*Procedure:* Return the current directory in a PDB file.

*Usage:* (current-directory file)

**current-pdbfile**
*C Information:* SX_current_file, SS_sargs
*Procedure:* Return the current default PDB file.

**def-common-types**
*C Information:* SX_def_common_types, SS_sargs
*Procedure:* Define some common internal SX data types to the given file.

**def-member**
*C Information:* Slist, UR_MACRO, SS_nargs
*Procedure:* Special Form: Create a member list.

**default-offset**
*C Information:* SX_default_offset, SS_nargs
*Procedure:* Set and/or return the default offset for the given file.

**defstr?**
*C Information:* SX_defp, SS_sargs
*Procedure:* Return #t if the object is a DEFSTR, and #f otherwise.

**desc-variable**
*C Information:* SX_desc_variable, SS_sargs
*Procedure:* Describe a PDB variable.

**diff-variable**
*C Information:* SX_diff_var, SS_nargs
*Procedure:* Compare an entry in two files

*Usage:* (diff-variable $file_1$ $file_2$ name [comparison-precision [display-mode]]).

**display-differences**
*C Information:* SX_display_diff, SS_nargs
*Procedure:* Display information returned by diff-variable.

**display-menu**
*C Information:* SX_menu, SS_nargs

*Procedure:* Display a menu of mappings, images, and curves.

**file-variable?**
*C Information:* SX_file_varp, SS_nargs
*Procedure:* Return #t if the variable is in the specified file, and #f otherwise.

*Usage:* (file-variable? file variable)

**find-types**
*C Information:* SS_find_types, SS_sargs
*Procedure:* Return a list of the derived data types in a variable.

**hash->pdbdata**
*C Information:* SX_hash_to_pdbdata, SS_nargs
*Procedure:* Convert a hash table to a PDBDATA object.

**indirection**
*C Information:* Slist, SS_nargs
*Procedure:* Create a type list.

**list-defstrs**
*C Information:* SX_list_defstrs, SS_sargs
*Procedure:* Return a list of the data types in a file.

**list-file**
*C Information:* SX_list_file, SS_zargs
*Procedure:* Return a list of open pdbfiles.

*Usage:* (list-file)

**list-variables**
*C Information:* SX_list_variables, SS_nargs
*Procedure:* Return a list of the variables in a file directory.

*Usage:* (list-variables file pattern type)

**major-order**
*C Information:* SX_major_order, SS_nargs
*Procedure:* Set and/or return the major order (row or column) for the given file.

**make-defstr**
*C Information:* SX_make_defstr, UR_MACRO, SS_nargs
*Procedure:* Special Form: Create a **DEFSTR** object from the list (macro version).

**make-defstr***
*C Information:* SX_make_defstr, SS_nargs
*Procedure:* Create a **DEFSTR** object from the list (procedure version).

**make-directory**
*C Information:* SX_make_directory, SS_nargs
*Procedure:* Create a directory in a PDB file.

*Usage:* (make-directory file directory)

**make-syment**
*C Information:* SX_make_syment, SS_nargs
*Procedure:* Create a **SYMENT** object from the list.

**open-pdbfile**
*C Information:* SX_open_pdbfile, SS_nargs
*Procedure:* Open and return a PDB file.

*Usage:* (open-pdbfile expression)

**pdb->list**
*C Information:* SX_pdb_to_list, SS_sargs
*Procedure:* Convert some PDB type to a list.

**pdb-read-numeric-data**
*C Information:* SX_read_numeric_data, SS_nargs
*Procedure:* Read a numeric array from a PDB file.

**pdbdata?**
*C Information:* SX_pdbdatap, SS_sargs
*Procedure:* Return #t if the object is a **PDBDATA**, and #f otherwise.

**pdbdata->hash**
*C Information:* SX_pdbdata_to_hash, SS_sargs
*Procedure:* Convert a **PDBDATA** object to a hash object.

**pdbfile?**
*C Information:* SX_pdbfp, SS_sargs
*Procedure:* Return #t if the object is a PDB file, and #f otherwise.

**set-format**
*C Information:* SX_set_format, SS_nargs
*Procedure:* Set a format printing control value. These values control the printed representation of numerical types.

*Usage:* (set-format format value)

**set-switch**
*C Information:* SX_set_switch, SS_nargs
*Procedure:* Set a switch which controls the behavior of SX. The switches are:

**0 --** structure printing prefix (0 -> full path, 1 -> space only, 2-> current leaf)

**1 --** structure printing information (0 -> print name only, 1-> print name and type)

**2 --** control recursive indentation (0 -> recurse and indent, 1-> print the recursion depth count but don't indent)

**3 --** maximum number items a variable can have before being printed in array format

**4 --** number of items printed on a line.

*Usage:* (set-switch switch value)

*Example:* (set-switch 3 10)

**show-pdb**
*C Information:* SX_show_pdb, SS_nargs
*Procedure:* Display the contents of a guest variable.

**syment?**
*C Information:* SX_symp, SS_sargs
*Procedure:* Return #t if the object is a **SYMENT**, and #f otherwise.

**target**

*C Information:* SX_target, SS_nargs
*Procedure:* Set and/or return the data standard and alignment to be applied to next file created.

**type**
*C Information:* SX_pdb_type, SS_nargs
*Procedure:* Create a type list for **write-pdbdata**.

**write-pdbdata**
*C Information:* SX_write_pdbdata, SS_nargs
*Procedure:* Write PDB data to a file and encapsulate it as a **PDBDATA** object.

**write-ultra-curve**
*C Information:* SX_wrt_ultra_curve, SS_nargs
*Procedure:* Create an ULTRA curve.

## Low Level Binary File Access Functions

**close-raw-binary-file**
*C Information:* SX_close_raw_file, SS_nargs
*Procedure:* Close a file opened with open-raw-binary-file.

> *Usage:* (close-raw-binary-file file)
>
> *Example:* (close-raw-binary-file file-obj)

**open-raw-binary-file**
*C Information:* SX_open_raw_file, SS_nargs
*Procedure:* Open an arbitrary file as a PDB file without a symbol table and possesing a minimal structure chart. Subsequent data access calls will provide the missing information. This file can be closed with **close-raw-binary-file**.

> *Usage:* (open-raw-binary-file name mode type $data\_standard_i$ $data\_alignment_i$)
>
> *Example:* (open-raw-binary-file "foo" "r" NETCDF 1 1)

**read-binary**
*C Information:* SX_rd_raw, SS_nargs
*Procedure:* Read arbitrary binary data from file using the specified information and return it as a SCHEME object. This is a low level access mechanism. See **read-pdbdata** for the main high level PDB interface call.

> *Usage:* (read-binary file address nitems file-type [mem-type])

*Example:* (read-binary data 10 1 "integer")

```
(read-binary data 14 5 "integer" "long")
```

**write-binary**
*C Information:* SX_wr_raw, SS_nargs
*Procedure:* Write arbitrary binary data encapsulated as a SCHEME object to a file using the specified information. This is a low level access mechanism. See **write-pdbdata** for the main high level PDB inteface call.

*Usage:* (write-binary file object address nitems file-type [mem-type])

*Example:* (write-binary data 3 10 1 "integer")

**PGS Functions**

The majority of the PGS functions are documented in the PGS User's Manual. You are encouraged to read that manual since it explains not only the SX bindings of the graphical functionality but the drawing model and general approach used in PGS.

The functions which are documented here are ones that have less to do with the PGS API than the usage of graphical objects in the SX context. For example, the predicates for the PGS objects are irrelevant to the PGS API but are an important part of using them in SX applications.

**pdbcurve->pg-graph**
*C Information:* SX_pdbcurve_graph, SS_nargs
*Procedure:* Read an ULTRA curve object from a PDB file.

*Usage:* (pdbcurve->pg-graph file name)

**pdbdata->pg-graph**
*C Information:* SX_pdbdata_graph, SS_nargs
*Procedure:* Read a PGS graph object from a PDB file.

*Usage:* (pdbdata->pg-graph file name)

**pdbdata->pg-image**
*C Information:* SX_pdbdata_image, SS_nargs
*Procedure:* Read a PGS image object from a PDB file.

*Usage:* (pdbdata->pg-image file name)

**pg-curve-name**
*C Information:* SX_get_ascii_curve_name, SS_nargs
*Procedure:* Return the curve referenced by name or menu number.

**pg-def-graph-file**
*C Information:* SX_def_file_graph, SS_sargs
*Procedure:* Set up a PDB file to receive PGS graph objects.

**pg-device?**
*C Information:* SX_devicep, SS_sargs
*Procedure:* Return #t if the object is a PGS device, and #f otherwise.

**pg-device-attributes?**
*C Information:* SX_dev_attributesp, SS_sargs
*Procedure:* Return #t if the object is a set of PGS device attributes, and #f otherwise.

**pg-graph?**
*C Information:* SX_graphp, SS_sargs
*Procedure:* Return #t if the object is a PGS graph, and #f otherwise.

**pg-graph->pdbcurve**
*C Information:* SX_graph_pdbcurve, SS_nargs
*Procedure:* Write an ULTRA curve object to a PDB file.

> *Usage:* (pg-graph->pdbcurve curve file)

**pg-graph->pdbdata**
*C Information:* SX_graph_pdbdata, SS_nargs
*Procedure:* Write a PGS graph object to a PDB file.

> *Usage:* (pg-graph->pdbdata graph file)

**pg-grotrian-graph?**
*C Information:* SX_grotrian_graphp, SS_sargs
*Procedure:* Return #t if the object is a PGS grotrian graph, and #f otherwise.

**pg-image?**
*C Information:* SX_imagep, SS_sargs
*Procedure:* Return #t if the object is a PGS image, and #f otherwise.

**pg-image->pdbdata**
*C Information:* SX_image_pdbdata, SS_nargs
*Procedure:* Write a PGS image object to a PDB file.

*Usage:* (pg-image->pdbdata image file)

**pg-image-name**
*C Information:* SX_get_ascii_image_name, SS_nargs
*Procedure:* Return the image referenced by name or menu number.

**pg-mapping-name**
*C Information:* SX_get_ascii_mapping_name, SS_nargs
*Procedure:* Return the mapping referenced by name or menu number.

**PANACEA Functions**

**pa-advance-name**
*C Information:* SX_advance_name, SS_sargs
*Procedure:* Advance the given file name according to the PANACEA naming convention. For a file, foo.xdd, where dd is a two digit base 36 number, the number dd is incremented.

**pa-advance-time**
*C Information:* SX_advance_time, SS_nargs
*Procedure:* Set the problem time and time step.

**pa-command**
*C Information:* SX_pan_cmmnd, SS_nargs
*Procedure:* Hand a command line off to PANACEA for processing by the installed commands.

**pa-define-variable**
*C Information:* SX_def_var, SS_sargs
*Procedure:* Define a new variable to the PANACEA database.

**pa-describe-entity**
*C Information:* SX_desc_pan, SS_sargs
*Procedure:* Display a description of a PANACEA variable or package.

**pa-display**
*C Information:* SX_display_pan_object, SS_sargs
*Procedure:* Display a PANACEA object in nice form.

**pa-finish-simulation**
*C Information:* SX_fin_system, SS_zargs
*Procedure:* Gracefully conclude a numerical simulation.

**pa-init-simulation**
*C Information:* SX_init_system, SS_zargs
*Procedure:* Initialize a numerical simulation.

**pa-install-commands**
*C Information:* SX_inst_com, SS_zargs
*Procedure:* Install the package generator commands so that they may be invoked by subsquent operations.

**pa-intern-packages**
*C Information:* SX_intern_packages, SS_zargs
*Procedure:* Return a list of variables which are bound to the PANACEA packages.

**pa-iv-specification?**
*C Information:* SX_iv_specp, SS_sargs
*Procedure:* Return #t if the object is a PANACEA initial value specification, and #f otherwise.

**pa-package?**
*C Information:* SX_packagep, SS_sargs
*Procedure:* Return #t if the object is a PANACEA package, and #f otherwise.

**pa-package-name**
*C Information:* SX_package_name, SS_sargs
*Procedure:* Return the name of the PANACEA package.

**pa-read-commands**
*C Information:* SX_readh, SS_nargs
*Procedure:* Read and process the user installed commands in the named file.

**pa-read-state-file**

*C Information:* SX_rd_restart, SS_nargs
*Procedure:* Read the named state file and do the specified conversions.

**pa-run-package**
*C Information:* SX_run_package, SS_sargs
*Procedure:* Execute the given package and return its time step and controlling zone.

**pa-save-to-pp**
*C Information:* SX_dump_pp, SS_zargs
*Procedure:* Save the data for the output requests from this cycle.

**pa-simulate**
*C Information:* SX_pan_simulate, SS_nargs
*Procedure:* Run a simulation from Ti to Tf.

**pa-source-variable?**
*C Information:* SX_srcvarp, SS_sargs
*Procedure:* Return #t if the object is a PANACEA source variable, and #f otherwise.

**pa-variable?**
*C Information:* SX_panvarp, SS_sargs
*Procedure:* Return #t if the object is a PANACEA variable, and #f otherwise.

**pa-variable->numeric-array**
*C Information:* SX_db_numeric_data, SS_sargs
*Procedure:* Save the data for the output requests from this cycle.

**pa-write-restart-dump**
*C Information:* SX_wr_restart, SS_zargs
*Procedure:* Write a restart dump.

## Variables

The following variables provide SX level access to compiled PACT variables.

**PANACEA Variables**

The following variables are used primarily in conjunction with the definition of PANACEA database variables. The following example illustrates their use:

```
(pa-define-variable "foo" "double" nil nil
        scope runtime class optional attribute
        number-of-zones dimension
        cm per eV units)
```

**dimension**
A string used to delimit the dimensions of variable definitions. Its value is "dimension". (Required)

**upper-lower**
A string used to indicate that the following two parameters in a dimension specification form a (min index, max index) pair. It's value is "upper-lower".

**offset-number**
A string used to indicate that the following two parameters in a dimension specification form a (offset, number of elements) pair. It's value is "offset-number".

**units**
An integer value used to denote the end of the unit specifications. Its value is -1.

**per**
An integer value used to denote the end of the numerator unit specifications. Its's value is -2.

**attribute**
An integer value used to denote the end of the variable attribute specifications. It's value is 102.

### Scope variables

**scope**
An integer used to denote that the next argument will specify the SCOPE attribute of the variable. It's value is 97.

**definition**
Denotes the DEFN SCOPE for the variable. It's value is -1.

**restart**
Denotes the RESTART SCOPE for the variable. It's value is -2.

**demand**
Denotes the DMND SCOPE for the variable. It's value is -3.

**runtime**
Denotes the RUNTIME SCOPE for the variable. It's value is -4.

**edit**
Denotes the EDIT SCOPE for the variable. It's value is -5.

**scratch**
Denotes the SCRATCH SCOPE for the variable. It's value is -6.

## Class variables

**class**
An integer used to denote that the next argument will specify the CLASS attribute of the variable. It's value is 98.

**required**
Denotes the REQU CLASS for the variable. It's value is 1.

**optional**
Denotes the OPTL CLASS for the variable. It's value is 2.

**pseudo**
Denotes the PSEUDO CLASS for the variable. It's value is 3.

## Persistence variables

**persist**
An integer used to denote that the next argument will specify the PERSIST attribute of the variable. It's value is 99.

**release**
Denotes the REL PERSIST for the variable. It's value is -10.

**keep**
Denotes the KEEP PERSIST for the variable. It's value is -11.

**cache**
Denotes the CACHE PERSIST for the variable. It's value is -12.

## Centering variables

**center**
An integer used to denote that the next argument will specify the CENTER attribute of the variable. It's value is 100.

**zone-centered**
Denotes the Z_CENT CENTER for the variable. It's value is -1.

**node-centered**
Denotes the N_CENT CENTER for the variable. It's value is -2.

**face-centered**
Denotes the F_CENT CENTER for the variable. It's value is -3.

**edge-centered**
Denotes the E_CENT CENTER for the variable. It's value is -4.

**uncentered**
Denotes the U_CENT CENTER for the variable. It's value is -5.

## Allocation variables

**allocation**
An integer used to denote that the next argument will specify the ALLOCATION attribute of the variable. It's value is 101.

**static**
Denotes the STATIC ALLOCATION for the variable. It's value is -100.

**dynamic**
Denotes the DYNAMIC ALLOCATION for the variable. It's value is -101.

## Units variables

The units variable here only reflect the default unit set of PANACEA. Applications are responsible for defining their own units with a compiled call such as:

**SS_install_cv("foo-unit",&foo_unit, SC_INTEGER_I);**

**radian**
An integer denoting the index of the RAD unit. Connected to PA_radian.

**steradian**
An integer denoting the index of the STER unit. Connected to PA_steradian.

**mole**
An integer denoting the index of the MOLE unit. Connected to PA_mole.

**Q**
An integer denoting the index of the Q unit. Connected to PA_electric_charge.

**cm**
An integer denoting the index of the CM unit. Connected to PA_cm.

**sec**
An integer denoting the index of the SEC unit. Connected to PA_sec.

**g**
An integer denoting the index of the G unit. Connected to PA_gram.

**eV**
An integer denoting the index of the EV unit. Connected to PA_eV.

**K**
An integer denoting the index of the K unit. Connected to PA_kelvin.

**erg**
An integer denoting the index of the ERG unit. Connected to PA_erg.

**cc**
An integer denoting the index of the CC unit. Connected to PA_cc.

# The SX Library

In this section we list some of the functions in the PACT SX library. Although there are many functions here, only a relative handful are useful to application developers wishing to integrate the interpreter into their product. All of the relevant declarations are made in the header file sx.h which should be #include'd in your source code.

## Constants

These #define'd constants are part of the essential glue of SX. These items by and large are the interface between the interpreter and the data structures of the other parts of PACT. See the discussion in the section Interfacing Compiled and Interpreted Code

### Object Type Constants

These constants define the type indeces for various PACT objects. They are used in conjunction with **SS_args, SS_call_scheme**, and **SS_make_list**. They are also implicitly used by the C and SCHEME level predicate functions.

| | | |
|---|---|---|
| G_FILE | 30 | SX generic file object type |
| G_DEFSTR | 31 | PDB defstr object type |
| G_SYMENT | 32 | PDB syment object type |
| G_DIMDES | 33 | PDB dimdes object type |
| G_MEMDES | 34 | PDB memdes object type |
| G_PDBDATA | 35 | SX pdbdata object type |
| G_PANVAR | 36 | PANACEA PA_variable object type |
| G_PACKAGE | 37 | PANACEA PA_package object type |
| G_SOURCE_VARIABLE | 38 | PANACEA PA_source_variable object type |
| G_IV_SPECIFICATION | 39 | PANACEA PA_iv_specification object type |
| G_PLOT_REQUEST | 40 | PANACEA PA_plot_request object type |
| G_GRAPH | 43 | PGS PG_graph object type |
| G_DEVICE | 44 | PGS PG_device object type |
| G_DEV_ATTRIBUTES | 45 | PGS PG_dev_attributes object type |
| G_NUM_ARRAY | 46 | PML C_array object type |
| G_MAPPING | 47 | PML PM_mapping object type |
| G_SET | 48 | PML PM_set object type |
| G_IMAGE | 50 | PGS PG_image object type |

## Macros

These macros facilitate interfacing with the interpreter and writing C level routines to be used by the interpreter.

### PDB Related Accessors

| | |
|---|---|
| FILE_FILE(type, x) | the file pointer cast to type * in the g_file object x |
| FILE_TYPE(x) | the type string from the g_file object x |
| FILE_EXT_TYPE(x) | the external type string from the g_file object x |
| FILE_NAME(x) | the name string from the g_file object x |
| FILE_STREAM(type, x) | the FILE * from the thing in the g_file object x |
| FILE_MENU(x) | the menu from the g_file object x |
| FILE_N_MENU_ITEMS(x) | the actual number of items in the menu |
| FILE_MAX_MENU_ITEMS(x) | the maximum number of items in the menu |
| PDBDATA_NAME(x) | the name string of a pdbdata object x |
| PDBDATA_DATA(x) | the data pointer of a pdbdata object x |
| PDBDATA_EP(x) | the symbol table entry of a pdbdata object x |
| PDBDATA_FILE(x) | the PDBfile of a pdbdata object x |

### Predicates

### PDBLib Types

| | |
|---|---|
| SX_DEFSTRP(x) | TRUE iff x is a defstr object |
| SX_SYMENTP(x) | TRUE iff x is a syment object |
| SX_PDBDATAP(x) | TRUE iff x is a pdbdata object |

### PANACEA Types

| | |
|---|---|
| SX_PANVARP(x) | TRUE iff x is a PA_variable object |
| SX_PACKAGEP(x) | TRUE iff x is a PA_package object |
| SX_SOURCE_VARIABLEP(x) | TRUE iff x is a PA_source_variabl |
| SX_IV_SPECIFICATIONP(x) | TRUE iff x is a PA_iv_specificatio |
| SX_PLOT_REQUESTP(x) | TRUE iff x is a PA_plot_reques |

### PGS Types

| | |
|---|---|
| SX_GRAPHP(x) | TRUE iff x is a PG_graph object |
| SX_IMAGEP(x) | TRUE iff x is a PG_image object |

| | |
|---|---|
| SX_DEVICEP(x) | TRUE iff x is a PG_device object |
| SX_DEV_ATTRIBUTESP(x) | TRUE iff x is a PG_dev_attributes object |

**PML Types**

| | |
|---|---|
| SX_NUMERIC_ARRAYP(x) | TRUE iff x is a C_array object |
| SX_MAPPINGP(x) | TRUE iff x is a PM_mapping object |
| SX_SETP(x) | TRUE iff x is a PM_set object |

## Variables

**Integers**

| | |
|---|---|
| SX_autorange | flag controlling auto ranging of plots |
| SX_autodomain | flag controlling auto domaining of plots |
| SX_autoplot | flag controlling replotting |
| SX_background_color_white | flag controlling white backgrounds for screens |
| SX_border_width | window border width in pixels |
| SX_cgm_background_color | flag controlling white backgrounds for CGM |
| SX_cgm_flag | CGM flag |
| SX_default_color | default color to use |
| SX_grid | flag controlling grids on plots |
| SX_label_length | number of label characters to print |
| SX_label_type_size | size in points of label font |
| SX_lines_page | number of lines per page in window |
| SX_plot_flag | flag controlling plottting |
| SX_plot_type_size | size in points of plot related fonts |
| SX_ps_flag | PostScript flag |
| SX_render_1d_1d | default 1d domain vs 1d range plot rendering |
| SX_render_2d_1d | default 2d domain vs 1d range plot rendering |
| SX_render_2d_2d | default 2d domain vs 2d range plot rendering |
| SX_show_mouse_location | flag controlling display of mouse location |
| SX_squeeze_labels | flag controlling whitespace compression in labels |
| SX_x_log_scale | flag controlling log scale in x direction (1d-1d only) |
| SX_y_log_scale | flag controlling log scale in y direction (1d-1d only) |

**REALs**

SX_botspace

SX_label_space

SX_leftspace

SX_marker_scale

SX_marker_orientation

SX_phi

SX_rightspace

SX_show_mouse_location_x

SX_show_mouse_location_y

SX_theta

SX_topspace

**Strings**

SX_cgm_name

SX_cgm_type

SX_palette

SX_plot_type_style

SX_ps_name

SX_ps_type

SX_smooth_method

## Functions

The following functions are just some of the many functions in the SX library. They are declared in sx.h.

**Input/Output**

These functions augment the I/O capabilities of the basic interpeter.

**object *SX_pp_names(object *argl)**

**void SX_parse(PFByte replot, PFPChar reproc)**

**Top Level**

void SX_init(char *code, char *vers)

Initialize the SX system. This includes all the initialization of the interpreter done with **SS_init**.

# Interfacing Compiled and Interpreted Code

In this section we will explain how to make your C (or Fortran) data and routines available to the SCHEME or SX interpreter (referred to simply as the interpreter from now on). We will also explain how to have your C or Fortran routines (referred to as compiled routines or compiled code) call interpreted functions (that is SCHEME code).

This information will enable developers to build knowledge of their compiled code into the interpreter so that they or their users can write interpreted code that uses the full power of SCHEME. SX itself is an example of this approach to development. SX is largely involved with defining the structures of PACT to the interpreter and supplying routines to manipulate them. As a result of this, PDBView is an SX program which manipulates binary data files in more general terms than any fixed set of compiled routines could do.

The other interesting capability which we will discuss is the ability to have compiled routines call interpreted ones. This component gives a complete interface by which compiled and interpreted routines interact. Developers then have maximal flexibility in using SCHEME or SX with their applications

## Some PACT SCHEME Implementation Details

We present here a discussion of details of the implementation of the interpreter which are necessary in order to be able to successfully interface your applications with the interpreter.

### Objects

All SCHEME objects carry certain information with themselves. In particular, all SCHEME objects know: what their data type is; how they are to be evaluated; how to print themselves; and how to release themselves. The object type is defined in the header file scheme.h.

With the exception of lists (actually CONS cells) and numbers all objects also have a printing name. This is the easiest way to handle the printing of objects in that the print name can be set when the object is created and a single printing method can be used for all objects with a non-NULL print name.

## C Level Macros

The following C macros (#define'd in scheme.h) define the application access to SCHEME objects:

**SS_INQUIRE_OBJECT(x)**
Return a pointer to the SCHEME object known to the interpreter's symbol table by the name given in x.

**SS_OBJECT_GC(x)**
Return the number of references to the given object.

**SS_UNCOLLECT(x)**
Render the memory management facility unable to release the given object.

**SS_GET(type, x)**
Return the value part of the given object (a pointer) cast to the specified type.

**SS_OBJECT_TYPE(x)**
Return the integer type value for the given object.

**SS_OBJECT_NAME(x)**
Return the print name of the given object.

**SS_OBJECT(x)**
Return a pointer to the value part of the given object.

**SS_OBJECT_ETYPE(x)**
Return the integer evaluation type for the given object.

**SS_OBJECT_PRINT(x, strm)**
Print the given object on the specified I/O stream. You should not have to use

this since the interpreter will use it on your behalf

**SS_OBJECT_FREE(x)**
Release the given object. You should not have to use this, the interpreter will invoke it when necessary.

## Evaluation Types

Every SCHEME object has an evaluation type. This tells the evaluator how to treat the object. There are three evaluation types (defined in scheme.h):

**SELF_EV**
Such objects evaluate to themselves (this is what most SCHEME objects do)

**VAR_EV**
Such objects are variables and evaluate to the value to which they are bound

**PROC_EV**
Such objects are evaluated as procedure calls (only the CONS should have this evaluation type!!!)

## SCHEME Function Protocol

PACT SCHEME has a non-standard mechanism for specifying how the interpreter will process the arguments to procedure calls and how the return value will be processed. This property is assigned to primitive (i.e. C level) procedures when they are installed in the SCHEME symbol table. This permits one C level procedure to be installed in more than one way (under different names) to achieve different effects. The following constants are #define'd in scheme.h.

**SS_PR_PROC**
Evaluate the arguments to the call and do not re-evaluate the result of the call

**SS_EE_MACRO**
Evaluate the arguments to the call and re-evaluate the result of the call

**SS_UR_MACRO**
Do not evaluate the arguments to the call and do not re-evaluate the result of the call

**SS_UE_MACRO**
Do not evaluate the arguments to the call and re-evaluate the result of the call

SCHEME procedures defined using **define** are SS_PR_PROC and macros defined using **define-macro** are SS_UR_MACRO.

To illustrate some of the protocol uses:

| | |
|---|---|
| **read** and **pair?** | are implemented as SS_PR_PROC |
| **let** and **let*** | as SS_UE_MACRO |
| **lambda** and **quote** | as SS_UR_MACRO |
| **apply** | as SS_EE_MACRO. |

A loose convention for functions installed as both macros and procedures is for the procedure name to be the same as the macro name only with a trailing '*' added. For example, foo and foo* would be the macro and procedure versions of some functionality called foo. In applications involving user interactions (SCHEME level read-eval-print loops) the macro form is generally to be preferred in that users generally type simple expressions at command line prompts and they generally don't want to deal with any more punctuation than absolutely necessary. By contrast, when users write their own functions they generally want the procedure form. Since these functions are prepared "one time" with a text editor, the extra syntax if any is tolerable.

## Function Handlers

To factor out commonality among different functions, the interpreter calls primitive procedures in a two step process. Each C level function is called by a C level function called a handler. It is the job of the handler to do all of the work generic to the class of functions for which the handler may be used. The function itself does the specific operations only. This approach guarantees that all of the functions invoked by a handler are uniformly treated. This lends a great deal of semantic consistency to applications. Application developers may supply both handlers and functions thus giving them great flexibility in designing their products.

For example, consider the case of the operators +, -, *, /. Only one handler is needed to process the argument lists which each of the operators would get. Each of the basic functions only need be written to operate on a pair of numbers and return a numeric result. It is the handler that performs the looping over the argument list, carries out type checking; and accumlates the result.

PACT SCHEME supplies three very fundamental handlers:

**SS_zargs** - which calls functions that take no arguments

**SS_sargs** - which calls functions which take a single argument

**SS_nargs** - which calls functions which take a list of arguments.

All handlers must return a pointer to an object. All handlers take two arguments. First is a pointer to the basic function. The return type is specified as pointer to an object. However, since the handler is responsible for the actual value returned to the interpreter, the function can be cast to any type which the handler wants to deal with. The second argument to a handler is a SCHEME list of the arguments supplied. There is always a list given to the handler even if it is empty. How the handler processes that list and how it invokes the function that is the first argument is entirely up to the application developer!

# Defining New Objects to the Interpreter

To fully exploit the PACT SCHEME interpreter as a user extensibility mechanism, you should consider defining your own data structures (C structs) to the interpreter so that they become (or can be treated as) true SCHEME objects. Once this is done, the interpreter can operate on these objects in the ways which you define, and your users can extend your application to match their requirements.

To define your data structures, supply the following code elements:

**1. Define an index value for your type (start at 1000 or more)**

**2. Define a C level predicate macro for this type (see ARRAYP below)**

**3. Define any useful structure accessor macros (see ARRAY_TYPE and others below)**

**4. Write three functions to: instantiate, release, and print the object.**

**5. Write a function implementing the SCHEME level predicate (see arrayp below)**

**6. Write any other functions and/or handlers to manipulate the object at the SCHEME level**

**7. Install the functions from steps 5 and 6<br>(see install_array_functions below)**

You are now able to have the interpreter understand your data structures well enough to perform whatever operations you have defined for them. By doing a careful layering of functionality, it is possible to write a very basic and thin interface between C and SCHEME level coding and develop more elaborate functionality entirely in SCHEME.

## Array Example

Here is an example illustrating the above procedure. This is very close to how SX handles the PML defined C_array structure (in fact this is a simplified version of that code).

```
#include

typedef struct s_array
   {long length;
    char *type;
    void *data; array;

/* this is the index from step #1 */
#define G_NUM_ARRAY         1000

/* this is the predicate mentioned in step #2 */
#define ARRAYP(obj)         (SS_OBJECT_TYPE(obj) == G_NUM_ARRAY)

/* these are macros suggested by step #3 */
#define ARRAY(obj)          SS_GET(array, obj)
#define ARRAY_TYPE(obj)     (SS_GET(array, obj)->type)
#define ARRAY_LENGTH(obj)   (SS_GET(array, obj)->length)
#define ARRAY_DATA(obj)     (SS_GET(array, obj)->data)

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* _WR_GNUM_ARRAY - print a g_num_array
 *                - this is the function invoked by SS_OBJECT_PRINT
 *                - this is part of step #4
 */

static void _wr_gnum_array(obj, strm)
   object *obj, *strm;
   {PRINT(SS_OUTSTREAM(strm), "", ARRAY_TYPE(obj));

    return;

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* _RL_GNUM_ARRAY - release a g_num_array
 *                - this is the function invoked by SS_OBJECT_RELEASE
 *                - this is part of step #4
 */

static void _rl_gnum_array(obj)
   object *obj;
   {array *arr;

    arr = ARRAY(obj);

    SFREE(arr->data);
    SFREE(arr);

    SS_rl_object(obj);

    return;

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* _MK_ARRAY - encapsulate an array as an object
 *           - this is part of step #4
 */
```

```
object *_mk_array(arr)
   array *arr;
   {object *op;

    op = SS_mk_object(arr, G_NUM_ARRAY, SELF_EV, arr->type);
    op->print   = _wr_gnum_array;
    op->release = _rl_gnum_array;;

    return(op);

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* ARRAYP - function version of ARRAYP macro
 *        - this is the function mentioned in step #5
 */

object *arrayp(obj)
   object *obj;
   {return(ARRAYP(obj) ? SS_t : SS_f);

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* MK_ARRAY - allocate and return a array
 *          - form: (make-array   )
 *          - this is a step #6 function
 */

static object *mk_array(argl)
   object *argl;
   {array *arr;
    long size, bpi;
    char *type, ltype[MAXLINE];

    type = NULL;
    size = 0L;
    SS_args(argl,
            SC_STRING_I, &type,
            SC_LONG_I, &size,
            SC_LONG_I, &bpi,
            0);

    arr = MAKE(array);

    sprintf(ltype, "%s *", type);

    arr->type   = SC_strsave(ltype);
    arr->length = size;
    arr->data   = (byte *) MAKE_N(char, size*bpi);

    return(_mk_array(arr));

/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

/* INSTALL_ARRAY_FUNCS - install some array extensions to SX
 *                     - this is step #7
 */

void install_array_funcs()
```

```
   {

/* install the fundamental array operations */
    SS_install("array?",
               "Return #t if the object is a numeric array, and #f otherwis
               SS_sargs, arrayp, SS_PR_PROC);

    SS_install("make-array",
               "Allocate and return an array of the specified type and size
               SS_nargs, mk_array, SS_PR_PROC);

/* suggestions for other SCHEME level operations with arrays */

#if 0
    SS_install("list->array",
               "Return a numeric array built from a list of numbers",
               SS_nargs, list_array, SS_PR_PROC);

    SS_install("array->list",
               "Return a list of numbers built from a numeric array",
               SS_sargs, array_list, SS_PR_PROC);

    SS_install("resize-array",
               "Reallocate the given array to the specified size",
               SS_nargs, resz_array, SS_PR_PROC);

    SS_install("array-ref",
               "Reference the nth element of an array",
               SS_nargs, array_ref, SS_PR_PROC);

    SS_install("array-set!",
               "Set the nth element of an array",
               SS_nargs, array_set, SS_PR_PROC);

    SS_install("array-length",
               "Return the length of the given numeric array",
               SS_sargs, num_arr_len, SS_PR_PROC);
#endif

    return;

/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/
```

# Compiling and Loading

To compile your C programs you must use one or the other (not both) of the following

**#include <scheme.h>**

**#include <sx.h>**

in the source files which deal with the interpreter.

To link your application you must use the following libraries in the order specified. For SCHEME only programs use:

**-lscheme -lppc -lpdb -lpml -lscore [-lm ...]**

For SX programs use:

**-lsx -lscheme -lpanacea -lpgs [-lX11] -lppc -lpdb -lpml -lscore [-lm ...]**

Although this is expressed as if for a UNIX linker, the order would be the same for any system with a single pass linker. The items in [] are optional or system dependent.

Each system has different naming conventions for its libraries and the reader is assumed to understand the appropriate naming conventions as well as knowing how to tell the linker to find the installed PACT libraries on each system that they use.

# Calling Interpreted Routines from Compiled Routines

The only real trick to having a compiled routine call an interpreted one is to map the compiled parameters or arguments into a representation which the evaluator is able to use. Naturally, that means making a list of SCHEME objects out of the arguments and applying the interpreted function to them. The interpreted function can be specified by name so that part it easy. To make the correct SCHEME objects (referred to simply as objects from now on - sorry but the LISP folks co-opted that term long before the OOP folks) from the compiled variables, you need only give the integer object type designation (see the previous section) and a pointer to the variable for each argument. The function **SS_call_scheme** does the rest. Specifically, it builds a list of objects from the supplied information, looks up the named procedure, applies it to the list of arguments, and returns the value which the interpreted routine returns. The important part to note here is that **SS_call_scheme** returns a SCHEME object! Note also the similarity between SS_call_scheme and **SS_args**. These two functions are complementary and realize the majority of the run time interface between interpreted and compiled functions.

Here is an example continuing the array example of the last section:

```
    object *rv;
    char *type;
    int n, bpi;

/* invoke the SCHEME level function create-array
 * this is the same as (create-array "float" 10 4)
 */
    type = "float";
    n    = 10;
    bpi  = sizeof(float);
    rv = SS_call_scheme("create-array",
                        SC_STRING_I, type,
                        SC_INTEGER_I, &n,
                        SC_INTEGER_I, &bpi,
                        0);
```

```
                        .
                        .
                        .
     long length;
     char *ntype;
     double *data;

     length = ARRAY_LENGTH(rv);
     ntype  = ARRAY_TYPE(rv);
     data   = ARRAY_DATA(rv);
```

The SCHEME level use of this example is limited only by the number of functions defined so far to manipulate such objects. Here is a fragment of SCHEME code showing their use:

```
(let* ((x (make-array "double" 10 8)))

    (if (array? x)

         (printf nil "We have an array: %s\n" x)))
```

If we assume the existence of the functions alluded to in install_array_funcs we can show a more useful example:

```
(let* ((x (make-array "double" 10 8)))

    (array-set! x 5 15.34)

    (examine x 5)


(define (examine x n)

   (cond ((and (pair? x) (< n (length x)))

           (printf nil "List element %d = %s\n" n (list-ref x n)))

          ((and (array? x) (< n (array-length x)))

           (printf nil "Array element %d = %s\n" n (array-ref x n)))))
```

# Related Documents

SX is one part of PACT, a set of tools for portable code development and visualization. Users of SX may be interested in the other parts of PACT especially since the various aspects of SX are derived from PACT. That is the graphics is handled by PGS; and the binary file handling is done by PDBLib.

The list of PACT Documents is:

| | | |
|---|---|---|
| PACT User's Guide | UCRL-MA-112087 | |
| SCORE User's Manual | UCRL-MA-108976 Rev.1 | |
| PPC User's Manual | UCRL-MA-108964 Rev.1 | |
| PML User's Manual | UCRL-MA-108965 Rev.1 | |
| PDBLib User's Manual | M-270 Rev.2 | |
| PGS User's Manual | UCRL-MA-108966 Rev.1 | |
| PANACEA User's Manual | M-276 Rev.2 | |
| ULTRA II User's Manual | UCRL-MA-108967 Rev.1 | |
| PDBDiff User's Manual | UCRL-MA-108975 Rev.1 | |
| PDBView User's Manual | UCRL-MA-108968 Rev.1 | |
| SX User's Manual | UCRL-MA-112315 | Current Document |

---

**Last modified: Fri Feb 25 16:08:25 PST 2000**

*Modified by: rodrigues2@llnl.gov*