The Sol Language Project: A Scheme Implementation

Richard W. DeVaul
Prof. John Maeda
Aesthetics and Computation Group, MIT Media Lab

Revision: 2.22

Abstract

The Sol project explores the relationship between language design and creative expression. The Sol language grew out of our desire to create a syntax for expressing computation elegant and expressive enough to be an artistic medium in its own right. Sol provides an extreemly powerful set-theory based typing system and features a lexically parallel structure, meaning that concurrent operations can be seen at a glance in Sol source code.

Sol is currently implemented as an embedded language in R4RS Scheme, although the core implementation is slowly migrating to an all-Java implementation based on the Kawa Scheme-to-Java compiler. As this process continues, more and more of this documentation will refer to Javadoc generated class documentation.

The LaTeX version of this document was generated from a SchemeWEB source file using the SchemeWEB program written by John Ramsdell. The HTML version was created from the LaTeX using the LaTeX2HTML package written by Nikos Drakos[1]. The HTML version contains source code formatting errors not found in the LaTeX version.

[1] http://www-dsed.llnl.gov/files/programs/unix/latex2html/manual/

Contents

Chapter 1

The Sol Programming Language

1.1 Introduction

Sol[1] extends the syntax of Scheme LISP with concepts borrowed from set and category theory, to create a language that is beautiful, powerful, and concise. Although inspired by abstract and formal ideas from mathematics, the intent of Sol is to provide means of expressing parallel and sequential computation that is practical and useful, as well as elegant and beautiful.

Sol is inspired by the idea that there is an underlying aesthetic to coding, that the beauty of algorithms (like the beauty mathematics) is a wonderful and real thing, but that the means of expression commonly available to programmers and computational designers is pretty ugly. There has long been the notion of a trade-off between the practicality of a programming language and it's elegance. Sol is an attempt to create a langauge which is both practical and beautiful, in which it is possible both to get the job done and to express the underlying beauty of the solution. Every programming language is a medium of expression — Sol is an attempt to create a medium as beautiful as the message.

This document describes both the Sol language and its implementation as an embedded language in R4RS Scheme, although the core implementation is slowly migrating to an all-Java implementation based on the Kawa Scheme-to-Java compiler. As this process continues, more and more of this documentation will refer to Javadoc generated class documentation. For more information on Scheme, see the MIT Scheme home page at http://www-swiss.ai.mit.edu/scheme-home.html.

[1] Sol stands for Set Operations Language. Sol should not be confused with SETL (SET Language), an earlier set-theory inspired programming language.

1.2 Sol Concepts

Just as lists are a natural representation for sequential operations, sets provide a natural syntax for expressing concurrent operations, *e.g.* evaluating a set of procedures in Sol results the parallel evaluation of each procedure. By combining sets[2] and lists, Sol provides an elegant syntax for computation capable of describing a wide range of procedural forms.

1.2.1 Lists

Lists are a natural representation for sequential operation, and form the basis of the LISP (short for LISt Processing) programming languages. Scheme LISP was chosen as the starting point for Sol, because Scheme LISP provides a particularly elegant and syntax for expressing sequential computation. Sol supports all R^4 Scheme constructs[3].

1.2.2 Sets

Like a Scheme list, a Sol set is a collection of things. Unlike a Scheme list, a Sol set can be either finite or infinite, ordered or unordered. Every Scheme list has an explicit external representation, but many Sol sets don't have one, either because they are infinite (the set of complex numbers) or are the result of set operations which don't reduce to an explicit set. Sol sets are hierarchical — just as `COMPLEX` is a subset of `NUMBER`, all Sol entities are also subsets of the `SOL` set.

Sol defines a special set of procedures called *set operators*. The process of applying a set operator (*operator for short*) to a set is called an *operation*, and every operation results in a new (possibly empty) set. Generally speaking, set operations do not alter the contents of existing sets.

Sol sets come in three basic flavors. There are the *core* sets, which form the basis of all other sets and are defined as part of the language environment. There are *fundamental* sets, a designation which includes the core sets, Java classes, and explicit user-defined sets. And there are *derived* sets, which are the result of set operations on fundamental or derived sets, *e.g.*`(union boolean integer)`. Sets and set operations provide Sol with a very powerful and flexible typing system, as well as *lexical parallelism* — one can see concurrency in a Sol program simply by examining the source code.

[2]The term "set" has very strict mathematical requirements (it is difficult to prove that something is really a set) so formally speaking we should refer to Sol sets as classes. However, the term "class" has another conventional meaning when discussing programming languages. To avoid confusion, I will use the term *set* to refer to the mathematical entity *class* and *class* in the object-oriented programming sense of class.

[3]Scheme is intended as the starting point, not a subset of the language. In the future Sol may be implemented as a stand-alone compiler or interpreter, in which case R^4 compatibility may be sacrificed for the sake of clarity and efficiency.

Core Sets

Core sets are so-called because they are part of the core Sol language specification, and represent the primitive Sol types, *e.g.* the set of integers, strings, lists, *etc.*. Every R^4 Scheme type has a corresponding Sol core set. In addition, there are a number of core sets relating to aspects of the Sol language not shared with Scheme (the set of sets, for instance). Core sets are special in that all other sets in some way derive from core sets, either by explicitly including members or through operations on other sets.

Fundamental Sets

This category includes core sets, Java classes, and and user-defined explicit sets. These sets are called fundamental because they can be expressed without resorting to set operations on other sets, *e.g.*(set ''foo'' 3.14159265 real (set 1 2 3)) which is explicit, <java.lang.String>, which is a Java class, or (union (set 1 2 3) (set 4 5 6)) which simplifies to (set 1 2 3 4 5 6).

Derived Sets

A derived set is a set resulting from one or more set operations on fundamental or derived sets which does not simplify to a fundamental set, *e.g.* (union (set 1 2 3) string) (the union of $\{123\}$ and the set of strings) or (difference real integer) (the open set resulting from $\Re - I$). Note that (union (differencereal integer) integer) simplifies to real and hence is a fundamental, not a derived set[4]

Data typing

Data type in Sol is expressed through set membership. Any set (or set expression) can be used to specify type. Compound objects, such as lists or vectors, can be described in terms of the type of their elements. This can be seen in the syntax of the Sol function, which allows the name and type of each parameter to be specified, as well as the type of the return value. This mechanism of type specification is very powerful, since type constraints may be as tight or loose as the programmer wishes, *e.g.* (function integer ((a real) (b string) (c (set 1 "help!" 3))) ...) specifies an integer-valued function of a real, a string and a member of the set (set 1 "help!" 3), but (function sol ((a number) (b sol)) ...) specifies a function of a number (may be real, complex, *etc.*) followed by any Sol object, returning any Sol object.

1.2.3 Categories

Sol categories provide many of the advantages of objects, but with Sol langage orientation towards concurancy through set operations. Rather than starting

[4]The current implementation of Sol does not perform this type of simplification automatically, so the set resulting from this statement may be considered a derived set.

from a formal description, let us consider the example of representing and manipulating a three dimensional vector.

Function vs. Object Oriented

In a functional language, a three dimensional vector of reals (three-vector for short) could be a data structure holding three real numbers $(x\ y\ z)$ and a collection of functions operating on vectors $((function1)\ (function2)\ \ldots)$. In an object oriented programming environment one might define a three-vector class, combining the coordinates (the member data) with the vector functions (the member methods), *e.g.* $(x\ y\ z\ (\mathit{method1})(\mathit{method2})\ \ldots)$. The main advantages of the object oreinted approach are data abstraction, encapsulation, and inheritance. However, object orientation imposes certain limitiations. For instance, when dealing with large numbers of instances, an array of records,

$$((x_1\ y_1\ z_1)\ (x_2\ y_2\ z_2)\ \ldots)\,, \tag{1.1}$$

may be significantly more compact than the coresponding array of objects, especially if a function table is present:

$$((x_1\ y_1\ z_1\ (\mathit{method1}_1)\ (\mathit{method2}_1)\ldots)\ (x_2\ y_2\ z_2\ (\mathit{method1}_1)\ (\mathit{method2}_1)\ \ldots)\ \ldots)\,. \tag{1.2}$$

More importantly, the data-procedure encapsulation present in object instances may make it slower to manipulate objects than a corresponding collection of records[5]. If many operations of this type are performed (such as translating or rotating every vector in an array) the overhead can be significant.

Category as Container Class

To avoid this problem, one might design a new type of object (a container class) which holds an array of instances and methods for manipulating objects in the array. If all of the methods of interests are in the function table of the container class, than only one dereference is needed per call. Further, some of these methods (call them *operators*) may operate on the array as a unit, exploiting efficiencies of parallel operation unavailable to the member methods of the individual objects. Replace the array with a set, and this type of container class is essentially a Sol category.

Sol Categories

A Sol category contains a (possibly empty) set, zero or more *member operators*, and zero or more member functions. A member operator is a procedure of at least two arguments (a set and a set element) which returns a new element of a new set. Category member operators can be applied in parallel to the member set, resulting in a new set.

[5]For instance, referencing member functions ofen requires two pointer derefernces: the first to get the location of the object and another to locate the function using the object's function table.

Sol categories are similar to classes in a conventional object oriented programming language, providing data/method encapsulation and a single-inheritance mechanism. However, Sol categories differ from classes in a fundamental way; Rather than being a mechanism for encapsulating a heterogeneous collection of data and methods (a class), a category combines a single member set with methods for operating on that set in parallel, as well as more general methods which may operate on any data.

Through the inheritance mechanism "abstract" categories (categories containing empty sets) may be defined, which other "instance" categories inherit operators and methods from to acquire. Abstract categories can also serve as a useful organizational tool by providing named collections of static functions, just like static classes in a conventional object-oriented language.

1.3 Formal Description

Since Sol is still in flux, there is no formal description of the language available at this time. Please see the revised^4 report on Scheme for a formal description of that language, and the informal documentation and source code presented here for a description of Sol.

Chapter 2

The R4RS Scheme Implementation

Sol is currently implemented as an embedded language in R4RS Scheme Lisp. This has the advantages of flexibility, portability (Scheme runs on many platforms), and rapid language development. The disadvantages include a (temporary) lack of true parallelism and some performance limitations. In order to address these shortcomings, a Java-centric implementation is in progress which will maintain portability, improve performance, and support true parallelism.

The rest of this document is devoted to the actual Scheme (Kawa) sourcode implementing Sol. Symbols are defined before they are used wherever possible for the sake of readability, but this is not a restriction imposed by the Scheme programming language. Since Sol is a work in progress, the version of the document you are reading may contain documentation inconsistancies or source code errors. Whenever possible, rely on the latest version (available at `http://acg.media.mit.edu/people/rich/research/sol/index.html`), and where there is a conflict between the internal (source code) and external documentation of a procedure or macro, rely on the internal documentation.

2.1 Implementation Specific Code

This section documents the implementation differences for three Scheme environments. These differences are primarily in macro definition syntax. The following command loads the appropriate code as determined by the makefile.

2.1.1 Kawa Implementation

Preamble

Define a portable `eval` variation which assumes the current environment. In Kawa, this is is equivalent to the standard `eval`.

```
(define sol-eval eval)
```

Since the current version of Kawa lacks `cons*`, I'm implementing it here for convenience.

```
(defmacro cons* args
  (letrec
      ((recursive-cons
        (lambda (list)
          (if (and (list? list) (null? (cdr list)))
              (car list)
              `(cons ,(car list) ,(recursive-cons (cdr list)))))))
    `(cons ,(car args) ,(recursive-cons (cdr args)))))
```

Prompt for interactive sessions – hacked from kawa source tree.

```
(define (sol-prompter port)
  (let ((state (input-port-read-state port)))
    (if (char=? state #\Newline)
        ""
        (string-append (if (char=? state #\Space)
                           "#|Sol:"
                           (string-append "#|" (make-string 1 state) "---:"))
                       (number->string (input-port-line-number port))
                       "|# "))))
```

Make it the default prompter.

```
(set-input-port-prompter! (current-input-port) sol-prompter)
```

Macros

Fundamental macro definitions, Kawa style.

function and synchronized-function The `function` and `synchronized-function` macros are a type-checking extension to the lambda expression. The `synchronized-function` macro is identical to the `function` macro except that it is used to designate critical code sections for multi-threaded applications.

The first step in creating the `function` and `synchronized-function` macros is to define utility functions for argument checking. The procedure `function-args->lambda-args` recursively strips out formal lambda-style arguments from Sol function-style arguments passed to the `function` and `synchronized-function` macros.

```
(define (function-args->lambda-args args)
                                        ; Check for empty-list
                                        ; termination condition
  (cond ((null? args)
```

```
                                          ; Return empty list
  '())

                                          ; Check to see if args is a
                                          ; pair (it may not be if
                                          ; function-args-¿lambda-args was
                                          ; previously called with a
                                          ; non-list pair, e.g. (a . b).

 ((pair? args)

                                          ; Get the first argument unit
                                          ; and bind it to arg.

  (let ((arg (car args)))

    (cons                 ; Return a list of...
     (if (symbol? arg)
                                          ; If it's a symbol (an
                                          ; un-typed argument) just pass
                                          ; it along.

         arg

                                          ; Otherwise pull the symbol
                                          ; out of the IN expression.

         (car arg))

                                          ; ... followed by a recursive
                                          ; call to yourself with the
                                          ; rest of the list.

     (function-args->lambda-args (cdr args)))))

                                          ; We've fallen down to the
                                          ; else clause, meaning that
                                          ; args is just a symbol.
                                          ; Return it.

 (else args)))
```

The procedure `function-args->error-check` does type checking by calling `in?` to verify the type expressions in the Sol function `args` list.

```
(define (function-args->error-check args)
```

```
                                                ; Check for empty-list
                                                ; termination condition.

  (cond ((null? args)
                                                ; Return empty list.
         '())

                                                ; Check to see if args is a
                                                ; pair.

        ((pair? args)
                                                ; Get the first argument unit
                                                ; and bind it to arg.

         (let ((arg (car args)))

                                                ; If it's a symbol (an
                                                ; un-typed argument)...

           (if (symbol? arg)

                                                ; ... ignore it, and check the
                                                ; rest of the list.

               (function-args->error-check (cdr args))

                                                ; Otherwise, generate the
                                                ; following code, which is a
                                                ; recursive construction of
                                                ; type checks:

               `(if (not ,(cons 'in? arg))
                    (error "Bad argument to function:"
                           ',arg ,(car arg))
                    ,(function-args->error-check (cdr args))))))

                                                ; Args is a symbol – ignore it.
        (else '())))
```

Now we define the macros.

```
(defmacro function (type args . body)
```

```
                                         ; The body of the macro
                                         ; produces a lambda-expression
                                         ; followed by the
                                         ; type-checking code.

  `(lambda ,(function-args->lambda-args args)
     (let ((rval
            (call-with-current-continuation
             (lambda (return)
               ,(function-args->error-check args)
               ,@body))))
       (if (in? rval ,type)
           rval
           (error "function: Bad return value."
                  (list 'in? rval ,type) '-> '#f))))

  )
```

```
(defmacro synchronized-function (type args . body)

                                         ; The body of the macro
                                         ; produces a lambda-expression
                                         ; followed by the
                                         ; type-checking code.

  `(lambda ,(function-args->lambda-args args)
     (let ((rval
            (call-with-current-continuation
             (lambda (return)
               ,(function-args->error-check args)
               (synchronize
                (lambda ()
                  ,@body))))))
       (if (in? rval ,type)
           rval
           (error "synchronized-function: Bad return value."
                  (list 'in? rval ,type) '-> '#f))))

  )
```

operator and synchronized-operator The `operator` macro creates a function with three implicit arguments: A set and an element of that set. Additional arguments may be specified in the ordinary way. It returns zero or more new elemenents contained in a new set.

```
(defmacro operator (args . body)
```

```
  `(function sset
            ((this-set sset)
             this-element
             (this-index integer) ,@args) ,@body))
```

The `synchronized-operator` is the operator macro counterpart to the `synchronized-function`. Right now just make it an alias for `operator`.

```
(defmacro synchronized-operator (args . body)
  `(synchronized-function sset
                          ((this-set sset)
                           this-element
                           (this-index integer) ,@args) ,@body))
```

The `operator` macro creates a function with three implicit arguments: A set and an element of that set. Additional arguments may be specified in the ordinary way. It returns zero or more new elemenents contained in a new set.

(defmacro operator (type args . body) (if (in? sset type) ‘(function sset ((this-set sset) this-element (this-index integer) ,@args) ,@body) (error "operator: Must return a set." (list in? sset ,type) '-¿ '#f)))

Sol Category definitions

The `category` macro takes as its arguments a set and between zero and two lists of `letrec` style local bindings. The first list is a list of member operators, and the second is a list of member functions. The `category` macro returns a Sol category.

Categories use a simple single-inheritance mechanism in which all inherited members are "virtual." Eventually a more complex inheritance mechanism may be implemented if experience proves it necessary.

```
(defmacro category (super s ops . funcs)
  (letrec (
```

; Inside the body of the
; macro, all arguments are
; unevaluated. In order to
; work with the super-class in
; this environment we must
; force the evaluation of the
; super argument to get the
; category (procedure) it
; refers to.

```
           (parent-category (eval super))
```

; Force evaluation of S in

```
                                    ; order to perform type
                                    ; checking – S could be a
                                    ; procedure which returns a
                                    ; set rather than being a set.

(this-set (eval s))

                                    ; extract the operators and
                                    ; functions of the parent
                                    ; category.

(parent-operators (if (null? parent-category)
                      '()
                      ((parent-category 'get-operators))))

(parent-functions (if (null? parent-category)
                      '()
                      ((parent-category 'get-functions))))

                                    ; Construct the operator and
                                    ; function lists for this
                                    ; category.

(operators (append ops parent-operators))

                                    ; Note that FUNTIONS does not
                                    ; include the
                                    ; UNIVERSAL-FUNCTIONS defined
                                    ; above.

(functions (if (null? funcs)
               parent-functions
               (append (car funcs) parent-functions)))

                                    ; These functions are
                                    ; available to all Sol
                                    ; categories.
(universal-functions
 `((get-set (lambda () member-set))
   (set-set! (function
              sol ((new-set sset)) (set! member-set new-set)))
   (get-operators (lambda () ',operators))
   (get-operator-table (lambda () operator-table))
   (get-functions (lambda () ',functions))
   (get-function-table (lambda () function-table))))
```

```
                                        ; The construct-table
                                        ; procedure builds the
                                        ; operator and function
                                        ; tables.

     (construct-table
      (lambda (raw-table)
        (if (null? raw-table)
            '()

                                        ; These symbols already bound
                                        ; when construct-table is
                                        ; called.

            `(cons (list ',(caar raw-table)
                         ,(caar raw-table))
                   ,(construct-table (cdr raw-table))))))

                                        ; Check to see if the macro
                                        ; arguments were of the
                                        ; appropriate type.

(if (and (set? this-set) (list? ops))

                                        ; If they were, produce the
                                        ; following quasi-quoted code
                                        ; block. NOTE: This has to be
                                        ; quasi-quoted to do the
                                        ; appropriate substitutions
                                        ; with the unquote-operator.

   `(letrec
        ((member-set ,this-set)
         (member-operators ',operators)
         (member-functions ',functions)

                                        ; Declare the operators and
                                        ; functions so that they are
                                        ; part of the local name space.

                                        ; Note: Need to reverse these,
                                        ; otherwise subcategories
                                        ; can't override.

         ,@(reverse operators)
         ,@(reverse (append universal-functions functions))
```

```
      (function
       sol ((arg symbol))
       (letrec
           (
                                       ; create a table of member
                                       ; operators

            (operator-table ,(construct-table operators))

                                       ; Create a table of member
                                       ; functions.

            (function-table ,(construct-table
                               (append
                                universal-functions
                                functions)))

                                       ; Look up ARG in both tables.

            (operator-match (assq arg operator-table))
            (function-match (assq arg function-table)))

                                       ; If MATCH was found in
                                       ; operator-table...

         (if (pair? operator-match)
                                       ; Return lambda expression
             (lambda args
               (apply operate
                      (cons* member-set
                             (cadr operator-match) args)))

                                       ; else try function table...
             (if (pair? function-match)
                                       ; Return match
                 (cadr function-match)

                 (error "Unable to find method or field:" arg)
                 )))))

   (error '((s sset) (ops list)) (list s ops)))))
```

;(load "SolMacros")

2.1.2 Kawa Implementation

Preamble

Define a portable `eval` variation which assumes the current environment. In Kawa, this is is equivalent to the standard `eval`.

```
(define sol-eval eval)
```

Since the current version of Kawa lacks `cons*`, I'm implementing it here for convenience.

```
(defmacro cons* args
  (letrec
      ((recursive-cons
        (lambda (list)
          (if (and (list? list) (null? (cdr list)))
              (car list)
              `(cons ,(car list) ,(recursive-cons (cdr list)))))))
    `(cons ,(car args) ,(recursive-cons (cdr args)))))
```

Prompt for interactive sessions – hacked from kawa source tree.

```
(define (sol-prompter port)
  (let ((state (input-port-read-state port)))
    (if (char=? state #\Newline)
        ""
        (string-append (if (char=? state #\Space)
                           "#|Sol:"
                           (string-append "#|" (make-string 1 state) "---:"))
                       (number->string (input-port-line-number port))
                       "|# "))))
```

Make it the default prompter.

```
(set-input-port-prompter! (current-input-port) sol-prompter)
```

Macros

Fundamental macro definitions, Kawa style.

function and synchronized-function The `function` and `synchronized-function` macros are a type-checking extension to the lambda expression. The `synchronized-function` macro is identical to the `function` macro except that it is used to designate critical code sections for multi-threaded applications.

The first step in creating the `function` and `synchronized-function` macros is to define utility functions for argument checking. The procedure `function-args->lambda-args` recursively strips out formal lambda-style arguments from Sol function-style arguments passed to the `function` and `synchronized-function` macros.

```
(define (function-args->lambda-args args)
                                            ; Check for empty-list
                                            ; termination condition
  (cond ((null? args)
                                            ; Return empty list
         '())

                                            ; Check to see if args is a
                                            ; pair (it may not be if
                                            ; function-args-¿lambda-args was
                                            ; previously called with a
                                            ; non-list pair, e.g. (a . b).

        ((pair? args)

                                            ; Get the first argument unit
                                            ; and bind it to arg.

         (let ((arg (car args)))

           (cons                   ; Return a list of...
            (if (symbol? arg)
                                            ; If it's a symbol (an
                                            ; un-typed argument) just pass
                                            ; it along.

                arg

                                            ; Otherwise pull the symbol
                                            ; out of the IN expression.

                (car arg))

                                            ; ... followed by a recursive
                                            ; call to yourself with the
                                            ; rest of the list.

            (function-args->lambda-args (cdr args)))))

                                            ; We've fallen down to the
                                            ; else clause, meaning that
                                            ; args is just a symbol.
                                            ; Return it.

        (else args)))
```

The procedure `function-args->error-check` does type checking by calling `in?` to verify the type expressions in the Sol function `args` list.

```
(define (function-args->error-check args)

                                              ; Check for empty-list
                                              ; termination condition.

  (cond ((null? args)

                                              ; Return empty list.
         '())

                                              ; Check to see if args is a
                                              ; pair.

        ((pair? args)
                                              ; Get the first argument unit
                                              ; and bind it to arg.

         (let ((arg (car args)))

                                              ; If it's a symbol (an
                                              ; un-typed argument)...

           (if (symbol? arg)

                                              ; ... ignore it, and check the
                                              ; rest of the list.

               (function-args->error-check (cdr args))

                                              ; Otherwise, generate the
                                              ; following code, which is a
                                              ; recursive construction of
                                              ; type checks:

               `(if (not ,(cons 'in? arg))
                    (error "Bad argument to function:"
                           ',arg ,(car arg))
                    ,(function-args->error-check (cdr args))))))

                                              ; Args is a symbol – ignore it.
        (else '())))
```

Now we define the macros.

```
(defmacro function (type args . body)

                                          ; The body of the macro
                                          ; produces a lambda-expression
                                          ; followed by the
                                          ; type-checking code.

  `(lambda ,(function-args->lambda-args args)
     (let ((rval
            (call-with-current-continuation
             (lambda (return)
               ,(function-args->error-check args)
               ,@body))))
       (if (in? rval ,type)
           rval
           (error "function: Bad return value."
                  (list 'in? rval ,type) '-> '#f))))

  )

(defmacro synchronized-function (type args . body)

                                          ; The body of the macro
                                          ; produces a lambda-expression
                                          ; followed by the
                                          ; type-checking code.

  `(lambda ,(function-args->lambda-args args)
     (let ((rval
            (call-with-current-continuation
             (lambda (return)
               ,(function-args->error-check args)
               (synchronize
                (lambda ()
                  ,@body))))))
       (if (in? rval ,type)
           rval
           (error "synchronized-function: Bad return value."
                  (list 'in? rval ,type) '-> '#f))))

  )
```

operator and synchronized-operator The `operator` macro creates a function with three implicit arguments: A set and an element of that set. Additional arguments may be specified in the ordinary way. It returns zero or more new elemenents contained in a new set.

```
(defmacro operator (args . body)
  `(function sset
             ((this-set sset)
              this-element
              (this-index integer) ,@args) ,@body))
```

The `synchronized-operator` is the operator macro counterpart to the `synchronized-function`. Right now just make it an alias for `operator`.

```
(defmacro synchronized-operator (args . body)
  `(synchronized-function sset
                          ((this-set sset)
                           this-element
                           (this-index integer) ,@args) ,@body))
```

The `operator` macro creates a function with three implicit arguments: A set and an element of that set. Additional arguments may be specified in the ordinary way. It returns zero or more new elemenents contained in a new set.

(defmacro operator (type args . body) (if (in? sset type) '(function sset ((this-set sset) this-element (this-index integer) ,@args) ,@body) (error "operator: Must return a set." (list in? sset ,type) '-¿ '#f)))

Sol Category definitions

The `category` macro takes as its arguments a set and between zero and two lists of `letrec` style local bindings. The first list is a list of member operators, and the second is a list of member functions. The `category` macro returns a Sol category.

Categories use a simple single-inheritance mechanism in which all inherited members are "virtual." Eventually a more complex inheritance mechanism may be implemented if experience proves it necessary.

```
(defmacro category (super s ops . funcs)
  (letrec (

                                             ; Inside the body of the
                                             ; macro, all arguments are
                                             ; unevaluated. In order to
                                             ; work with the super-class in
                                             ; this environment we must
                                             ; force the evaluation of the
                                             ; super argument to get the
                                             ; category (procedure) it
                                             ; refers to.
```

```
(parent-category (eval super))

                                ; Force evaluation of S in
                                ; order to perform type
                                ; checking - S could be a
                                ; procedure which returns a
                                ; set rather than being a set.

(this-set (eval s))

                                ; extract the operators and
                                ; functions of the parent
                                ; category.

(parent-operators (if (null? parent-category)
                      '()
                      ((parent-category 'get-operators))))

(parent-functions (if (null? parent-category)
                      '()
                      ((parent-category 'get-functions))))

                                ; Construct the operator and
                                ; function lists for this
                                ; category.

(operators (append ops parent-operators))

                                ; Note that FUNTIONS does not
                                ; include the
                                ; UNIVERSAL-FUNCTIONS defined
                                ; above.

(functions (if (null? funcs)
               parent-functions
               (append (car funcs) parent-functions)))

                                ; These functions are
                                ; available to all Sol
                                ; categories.
(universal-functions
 `((get-set (lambda () member-set))
   (set-set! (function
              sol ((new-set sset)) (set! member-set new-set)))
   (get-operators (lambda () ',operators))
```

```
          (get-operator-table (lambda () operator-table))
          (get-functions (lambda () ',functions))
          (get-function-table (lambda () function-table))))

                                      ; The construct-table
                                      ; procedure builds the
                                      ; operator and function
                                      ; tables.

       (construct-table
        (lambda (raw-table)
          (if (null? raw-table)
              '()

                                      ; These symbols already bound
                                      ; when construct-table is
                                      ; called.

              `(cons (list ',(caar raw-table)
                           ,(caar raw-table))
                     ,(construct-table (cdr raw-table))))))

                                      ; Check to see if the macro
                                      ; arguments were of the
                                      ; appropriate type.

(if (and (set? this-set) (list? ops))

                                      ; If they were, produce the
                                      ; following quasi-quoted code
                                      ; block. NOTE: This has to be
                                      ; quasi-quoted to do the
                                      ; appropriate substitutions
                                      ; with the unquote-operator.

    `(letrec
         ((member-set ,this-set)
          (member-operators ',operators)
          (member-functions ',functions)

                                      ; Declare the operators and
                                      ; functions so that they are
                                      ; part of the local name space.

                                      ; Note: Need to reverse these,
                                      ; otherwise subcategories
```

```
                                ; can't override.

   ,@(reverse operators)
   ,@(reverse (append universal-functions functions)))

(function
 sol ((arg symbol))
 (letrec
     (

                                ; create a table of member
                                ; operators

      (operator-table ,(construct-table operators))

                                ; Create a table of member
                                ; functions.

      (function-table ,(construct-table
                        (append
                         universal-functions
                         functions)))

                                ; Look up ARG in both tables.

      (operator-match (assq arg operator-table))
      (function-match (assq arg function-table)))

                                ; If MATCH was found in
                                ; operator-table...

   (if (pair? operator-match)
                                ; Return lambda expression
       (lambda args
         (apply operate
                (cons* member-set
                       (cadr operator-match) args)))

                                ; else try function table...
       (if (pair? function-match)
                                ; Return match
           (cadr function-match)

           (error "Unable to find method or field:" arg)
           )))))
```

```
(error '((s sset) (ops list)) (list s ops)))))
```

2.2 Sol Sets

Sol objects are typed by their membership in sets. All primitive types are members of the core Sol sets, and all other sets ultimately derive from operations on or elements from core sets.

2.2.1 The Sol Set

The `SOL` set is the top set — every Sol set is a sub-set of `SOL`. It is a special entity containing itself, all other sets and the members of those sets, and by extension every possible expression in the Sol programming language. Every Sol set is a subset of `SOL`, and no other set contains it..

```
(define sol 'sol)                    ; meta, unordered
```

2.2.2 The Core Sets

The core sets form the basis of the Sol typing system.

The Set Container Sets

The container sets are so-called because they are sets of sets. Note that none of the container sets contain the `SOL` set.

The set of sets, unordered sets, and ordered sets. The set of sets is named `SSET` and `SORDERED` in order to avoid conflicting with the procedures `set` and `ordered`.

```
(define sset 'sset) (define sordered 'sordered)
(define unordered 'unordered)
```

Fundamental, core, explicit, and derived. These sets make further distinctions among sets. Note that `FUNDAMENTAL` and `CORE` contain themselves.

```
(define fundamental 'fundamental)    ; meta, unordered
(define core 'core)                  ; meta, unordered
(define explicit 'explicit)          ; unordered
(define derived 'derived)            ; unordered
```

Numbers

The Scheme numeric tower is a hierarchy of sets:

```
(define number 'number)              ; ordered
(define complex 'complex)            ; ordered
(define real 'real)                  ; ordered
(define rational 'rational)          ; ordered
(define integer 'integer)            ; ordered
```

Strings, Symbols, and Characters

Note that these are distinct entities in Sol, just as they are in Scheme.

```
(define sstring 'sstring)              ; ordered
(define symbol 'symbol)                ; ordered
(define char 'char)                    ; ordered
```

Pairs and Lists

The set of pairs contains the set of lists. Note that the set of lists is named `slist` to avoid conflicting with the Scheme procedure `list`.

```
(define pair 'pair)                    ; unordered
(define slist 'slist)                  ; unordered
```

Procedures

The set of procedures is the set of Scheme lambda expressions. Note that `function` and `operator` return members of this set.

```
(define procedure 'procedure)          ; unordered
```

Boolean

The set of boolean values: true, false.

```
(define boolean 'boolean)              ; unordered
```

2.2.3 Explicit Sets

Unordered Sets

set In this implementation an unordered explicit finite set is a list with 'SET as its `car`. A Sol set evaluates (in the sense of `eval`) to itself.

```
(define set
  (lambda a
    (cons 'set a)))
```

Ordered sets

ordered In this implementation an ordered user-defined set is a list with `'ordered` as its first element and a vector as its `cadr`. A Sol ordered set evaluates (in the sense of `eval`) to itself.

```
(define (ordered . a)
  (if (null? a)
      (list 'ordered (vector))
      (let ((vect (car a)))
        (if (vector? vect)
            (list 'ordered vect )
            (list 'ordered (list->vector a))))))
```

void This is the set containing `#!void`, the return value of procedures which return nothing else.

```
(define void (set #!void))
```

ordered-length A routine to return the length of an ordered set.

```
(define ordered-length
  (function integer
            ((a sordered))
            (vector-length (cadr a))))
```

construct-ordered The `construct-orderd` operation constructs an ordered set using a constructor function[1].

```
(define make-initialized-vector
  (lambda (k init)
    (letrec ((internal
              (lambda (vector index)
                (if (>= index k)
                    vector
                    (begin
                      (vector-set! vector index
                                   (apply init (list index)))
                      (internal vector (+ index 1)))))))
      (internal (make-vector k) 0))))

(define construct-ordered
  (function
   sordered
   ((size integer) (constructor procedure))
   (list 'ordered (make-initialized-vector
                   size
                   constructor))))
```

[1]Should `make-initialized-vector` really be defined here? I think this is defined in MIT scheme, if not in R4RS. Fri Jun 12 20:50:09 CDT 1998

2.2.4 Derived Sets

Derived sets in Sol result from the application of primitive set operations (union, intersection, difference) to fundamental or derived sets, the results of which do not simplify to a fundamental set.

Primative Set Operations

The primative set operations provide the basic means of manipulating sets. All of these operations take as arguments two (or in the case of `cross`, two or more) sets and return a new set.

auto-simplify The auto-simplify symbol defines the default simplification level for the primitive set operatiors. This is kind of a hack – I should probably prevent simplification all together and define an expicit `simplify` operator but it will be fast to add this test to existing code.

```
(define global-auto-simplify 0)
```

union This is a binary operation producing the union of two sets. `union` returns false if both objects are not sets.

```
(define metaset '(union sset (set sol)))

(define union
  (function metaset ((obj1 metaset)
                     (obj2 metaset))
    (letrec
```

; Procedure to compute the union of two explicit sets, (represnted as
; lists) and return the result (also represented as a list). Because
; this code is called for both ordered and unordered sets, it should
; maintain ordering (as much as possible).

; Actually above documentation is not true. Procedure calculates
; b-minus-a, which can be used to describe union in combination with
; original set B

```
        ((union-explicit-simplify
          (lambda (a b b-minus-a)
```

; A and B are lists containing
; the members of the input
; sets, and UNION-A-B is a
; list containing the elements
; of the union of the
; two sets. A should be no

```
                                  ; larger than B for optimal
                                  ; efficiency.

      (if (equal? a '())          ; Is A empty?
                                  ; We're done - return union.
                                  ; Reverse is called to maintain
                                  ; order.

        (reverse b-minus-a)

                                  ; A not empty. Check to see if
                                  ; (car A) in B...

        (if (member (car a) b)

                                  ; ...it is. That means we
                                  ; don't need to add it to
                                  ; B-MINUS-A.

            (union-explicit-simplify (cdr a) b b-minus-a)

                                  ; It's not in B. Add it to
                                  ; B-MINUS-A.

            (union-explicit-simplify
             (cdr a) b (cons (car a) b-minus-a))))))

; This function calculates the union of a core and an explicit
; set. Every member of the explicit set is checked for membership in
; the core set.

    (union-core-explicit
     (lambda (a b b-minus-a)

                                  ; A is the symbolic name of
                                  ; the core set. B is
                                  ; the derived set represented
                                  ; as a list. This function
                                  ; returns a list of those
                                  ; elments of B not in A.

      (if (equal? b '())          ; Is B empty? We're done.

        (reverse b-minus-a)

                                  ; B not empty. Check to see
```

```
                                        ; if the first element of B is
                                        ; in A.

          (if (in? (car b) a)
                                        ; It is. Don't add it to the list.

              (union-core-explicit a (cdr b) b-minus-a)

                                        ; It's not. Add it to B-MINUS-A.

              (union-core-explicit
               a (cdr b) (cons (car b) b-minus-a)))))))

                                        ; *******************************
                                        ; Beginning of main UNION routine.
                                        ; *******************************

    (cond

                                        ; Check to see if either
                                        ; object is the SOL set. If
                                        ; so, return SOL.

     ((or (eq? obj1 sol) (eq? obj2 sol))
      'sol)

                                        ; See if obj1 and obj2 are the
                                        ; same thing. If so, return
                                        ; obj1.

     ((eq? obj1 obj2)
      obj1)

                                        ; Now check the first object
                                        ; against the core sets.

     ((core? obj1)
                                        ; OBJ1 is a member of CORE.

      (cond
       ((core? obj2)

;; We know that both OBJ1 and OBJ2 are core. This section is devoted
;; to case-by-case matching for simplification.

        (case obj1
                                        ; If OBJ1 is SSET
```

```
((sset)
 (case obj2

                             ; The union of SSET and any
                             ; other set-containing set is
                             ; SSET.

   ((sordered unordered
              fundamental core explicit derived) 'sset)
   (else (list 'union obj1 obj2))))

                             ; If OBJ1 is SORDERED
((sordered)
 (case obj2

                             ; The union of SSET or
                             ; UNORDERED and ORDERED
                             ; produces SSET.

   ((sset unordered) 'sset)
   (else (list 'union obj1 obj2))))

                             ; If OBJ1 is UNORDERED
((unordered)
 (case obj2

                             ; The union of SSET or ORDERED
                             ; and UNORDERED results in
                             ; SSET.

   ((sset sordered) 'sset)

                             ; The union of UNORDERED and
                             ; unordered core sets which do
                             ; not contian orderd sets
                             ; results in UNORDERED.

   ((pair slist
          procedure boolean)
    'unordered)
   (else (list 'union obj1 obj2))))

                             ; If OBJ1 is FUNDAMENTAL

((fundamental)
 (case obj2
```

```
                                ; The union of FUNDAMENTAL and
                                ; DERIVED is SSET.

   ((sset derived) 'sset)
   ((core explicit) 'fundamental)
   (else (list 'union obj1 obj2))))

                                ; OBJ1 is CORE

((core)
 (case obj2
   ((sset) 'sset)
   ((fundamental) 'fundamental)
                                ; The union of CORE and
                                ; EXPLICIT is DERIVED.
   ((explicit) 'fundamental)
   (else (list 'union obj1 obj2))))

((explicit)
 (case obj2
   ((sset) 'sset)
   ((fundamental) 'fundamental)
                                ; The union of CORE and
                                ; EXPLICIT is DERIVED.
   ((core) 'fundamental)
   (else (list 'union obj1 obj2))))

                                ; If OBJ1 is DERIVED
((derived)
 (case obj2
                                ; The union of FUNDAMENTAL and
                                ; DERIVED is SSET.

   ((sset fundamental) 'sset)
   (else (list 'union obj1 obj2))))

                                ; OBJ1 is NUMBER

((number)
 (case obj2
   ((complex real rational integer) 'number)
   (else (list 'union obj1 obj2))))

                                ; If OBJ1 is COMPLEX
((complex)
```

```
 (case obj2
   ((number) 'number)
   ((real rational integer) 'complex)
   (else (list 'union obj1 obj2))))

                              ; If OBJ1 is REAL
((real)
 (case obj2
   ((number) 'number)
   ((complex) 'complex)
   ((rational integer) 'real)
   (else (list 'union obj1 obj2))))

                              ; If OBJ1 is RATIONAL
((rational)
 (case obj2
   ((number) 'number)
   ((complex) 'complex)
   ((real) 'real)
   ((integer) 'rational)
   (else (list 'union obj1 obj2))))

                              ; If OBJ1 is INTEGER
((integer)
 (case obj2
   ((number) 'number)
   ((complex) 'complex)
   ((real) 'real)
   ((rational) 'rational)
   (else (list 'union obj1 obj2))))

                              ; If OBJ1 is SSTRING,
                              ; SYMBOL, or CHAR

((sstring symbol char)
 (list 'union obj1 obj2))

                              ; If OBJ1 is PAIR

((pair)
 (case obj2
   ((slist) 'pair)
   (else (list 'union obj1 obj2))))

                              ; If OBJ1 is SLIST
```

```
      ((slist)
       (case obj2
         ((pair) 'pair)
         (else (list 'union obj1 obj2))))

                                        ; If OBJ1 is PROCEDURE or BOOLEAN

      ((procedure boolean)
       (list 'union obj1 obj2))))
```

;; In this section we deal with the case of OBJ1 being a core set and
;; OBJ2 being an explicit set. Basically, we call union-core-explicit
;; and be done with it.

```
    ((explicit? obj2)

                                        ; OBJ1 is CORE, and OBJ2 is
                                        ; EXPLICIT.

     (let ((val2 (set->list obj2)))

                                        ; Call union-core-explicit.

       (let ((result (union-core-explicit obj1 val2 '())))

         (if (null? result)
             obj1

                                        ; If both are ordered...

             (if (and (ordered? obj1) (ordered? obj2))

                                        ; Return the symbolic union of
                                        ; obj2 (core) and the
                                        ; ordered results of calling
                                        ; union-core-explicit
                                        ; with obj2 and the contents
                                        ; of obj1.

                 (list 'union obj1
                       (list->ordered result))

                                        ; One or both are not ordered.
                                        ; Return the symbolic union of
                                        ; obj2 and the unordered
                                        ; results of calling
```

```
                                        ; union-core-explicit.

                    (list 'union obj1
                          (list->unordered result)))))))
```

```
;; In this section we deal with the case of OBJ1 being a core set and
;; OBJ2 being a derived set. This is kind of tricky, because the
;; simplification of some states are not clear. I will probably have
;; to rewrite it at some point.

         ((derived? obj2)

          (letrec ((operation (car obj2))
                   (operand1 (cadr obj2))
                   (operand2 (caddr obj2)))

              (cond
                                        ; If the operation is union...
               ((eq? operation 'union)
                (let ((result1 (union obj1 operand1))
                      (result2 (union obj1 operand2)))
                  (cond
                   ((core? result1)
                    (list 'union result1 operand2))
                   ((core? result2)
                    (list 'union operand1 result2))
                   (else
                    (list 'union obj1 obj2)))))

                                        ; Intersection...

               ((eq? operation 'intersection)
                (cond
                 ((subset? obj2 obj1)
                  obj1)
                 ((subset? obj1 obj2)
                  obj2)
                 (else
                  (list 'union obj1 obj2))))

                                        ; Difference.

               ((eq? operation 'difference)
                (cond
                 ((subset? obj2 obj1)
```

```
          obj1)
         ((subset? obj1 obj2)
          obj2)
         (else
          (list 'union obj1 obj2))))

       (else (error "union: Bad derived set:" obj2)))))
 ((classType? obj2)
  (list union obj1 obj2))
 (else (error "union: argument not a set: " obj2))))
```

;; We have now exhausted the possibilities for OBJ1 being a core set.
;; Moving on to the case where OBJ1 is an explicit set...

```
                                       ; Is object1 an explicit set?
((explicit? obj1)
                                       ; Is OBJ2 a core set?
 (cond
  ((core? obj2)

                                       ; We've already coded the
                                       ; union of a core and explicit
                                       ; set, so swap the order of
                                       ; the arguments and
                                       ; recursively call UNION.
   (union obj2 obj1))
```

;; Deal with the case of both OBJ1 and OBJ2 being explicit sets.

```
                                       ; Construct the union of two
                                       ; explicit sets.
  ((explicit? obj2)

                                       ; Convert both sets to lists.

   (let ((val1 (set->list obj1))
         (val2 (set->list obj2)))

                                       ; Are we simplifying?

     (if (> global-auto-simplify 0)
                                       ; If both sets are ordered

         (if (and (ordered? obj1) (ordered? obj2))
```

```
                                    ; Return the ordered union
           (list->ordered
            (append (union-explicit-simplify val1 val2 '()) val2))

                                    ; One or both of the sets is
                                    ; not an ordered set.  The
                                    ; result will not be ordered.

           (list->unordered
            (append (union-explicit-simplify val1 val2 '()) val2)))

                                    ; Same thing but no simplification.

       (if (and (ordered? obj1) (ordered? obj2))

                                    ; Return the ordered union
           (list->ordered
            (append val1 val2))

                                    ; One or both of the sets is
                                    ; not an ordered set.  The
                                    ; result will not be ordered.

           (list->unordered
            (append val1 val2))))))


;; The first object is explicit and the second object is neither core
;; nor explicit.  See if it's derived...

   ((derived? obj2)

    (letrec ((operation (car obj2))
             (operand1 (cadr obj2))
             (operand2 (caddr obj2)))

      (cond
                                    ; If the operation is union...
       ((eq? operation 'union)

                                    ; Subtract both operands from
                                    ; obj1 and see what we're left
                                    ; with.

        (let ((result (difference operand2
```

```
                                  (difference operand1 obj1))))

                                  ; If it's null, return OBJ2.

            (if (null? (set->list result))
                obj2
                                  ; Otherwise...
                (list 'union result obj2))))

                                  ; Intersection...

         ((eq? operation 'intersection)
          (cond
           ((subset? obj2 obj1)
            obj1)
           ((subset? obj1 obj2)
            obj2)
           (else
            (list 'union obj1 obj2))))

                                  ; Difference.

         ((eq? operation 'difference)
          (cond
           ((subset? obj2 obj1)
            obj1)
           ((subset? obj1 obj2)
            obj2)
           (else
            (list 'union obj1 obj2))))

         (else (error "union: Bad derived set: " obj2)))))
     ((classType? obj2)
      (list union obj1 obj2))
     (else (error "union: Second argument not a set: " obj2))))


;; We know that obj1 is neither core nor explicit. The only other
;; (legal) possibilities are that it is a derived or class type set.
;; Fortunately, we have coded for most of these combinations
;; already...

   ((fundamental? obj1)
    (cond
     ((core? obj2)
      (union obj2 obj1))
```

```
        ((explicit? obj2)
         (union obj2 obj1))

                                ; The only combination we have
                                ; left to wory about is the
                                ; union of a derived and a
                                ; derived set.

        ((derived? obj2)

                                ; There are nine special cases
                                ; to worry about. At this
                                ; point I'm just going to set
                                ; up the framework, then punt.

         (letrec ((op1 (car obj1))
                  (A (cadr obj1))
                  (B (caddr obj1))
                  (op2 (car obj2))
                  (C (cadr obj2))
                  (D (caddr obj2)))

                                ; Just return the unsimplified
                                ; expression.

           (list 'union obj1 obj2)))
        ((classType? obj2)
         (list union obj1 obj2))
        (else (error "union: second argument not a set: " obj2))))
      (else (error "union: first argument not a set: " obj1))))))
```

intersection `intersection` is a binary operation producing the intersection of two sets.

This implementation of intersection is incomplete, as it only simplifies in the case where both sets are explicit. A significant expansion will be required to get this working in the more general case.

```
(define intersection
  (function metaset ((obj1 metaset)
                     (obj2 metaset))
    (cond

                                ; If both objects are the
                                ; same, return the first set.
     ((eq? obj1 obj2)
      obj1)
```

```
                                         ; If either OBJ1 or OBJ2 are
                                         ; SOL, return the other
                                         ; object.

     ((eq? obj1 sol)
      obj2)

     ((eq? obj2 sol)
      obj1)

                                         ; Now check the first object
                                         ; against the core sets.

     ((core? obj1)
                                         ; OBJ1 is a member of CORE.
      (cond
       ((core? obj2)

;; We know that both OBJ1 and OBJ2 are core. This section is devoted
;; to case-by-case matching for simplification.

        (case obj1
          ((sset)
           (case obj2

                                         ; The intersection of SSET an
                                         ; any other set-containing set
                                         ; B is set B.

             ((sordered unordered
                        fundamental core explicit derived) obj2)
             (else (list 'intersection obj1 obj2))))

                                         ; If OBJ1 is SORDERED
          ((sordered)
           (case obj2

                                         ; The intersection of SSET and
                                         ; SORDERED results in SORDERED.

             ((sset) sordered)
             (else (list 'intersection obj1 obj2))))

          ((unordered)
           (case obj2
```

```
                                ; The intersection of SSET and
                                ; UNORDERED results in UNORDERED.

   ((sset) unordered)
   (else (list 'intersection obj1 obj2))))

                                ; If OBJ1 is FUNDAMENTAL

((fundamental)
 (case obj2

                                ; The intersection of FUNDAMENTAL and
                                ; DERIVED is empty.

    ((sset derived) '(set ))

                                ; CORE and EXPLICIT are
                                ; subsets of FUNDAMENTAL.

    ((core explicit) obj2)
    (else (list 'intersection obj1 obj2))))

                                ; OBJ1 is CORE

 ((core)
  (if (core? obj2)
      obj2
      (list 'intersection obj1 obj2)))

                                ; There are no CORE sets
                                ; (other than themselves)
                                ; which have non-empty
                                ; intersections with EXPLICIT
                                ; and DERIVED.

 ((explicit derived)
  '(set ))

                                ; If OBJ1 is NUMBER

 ((number)
  (case obj2
    ((complex real rational integer) obj2)
    (else (list 'intersection obj1 obj2))))
```

```
                                   ; If OBJ1 is COMPLEX
((complex)
 (case obj2
   ((number) complex)
   ((real rational integer) obj2)
   (else (list 'intersection obj1 obj2))))

                                   ; If OBJ1 is REAL
((real)
 (case obj2
   ((number complex) 'real)
   ((rational integer) obj2)
   (else (list 'intersection obj1 obj2))))

                                   ; If OBJ1 is RATIONAL
((rational)
 (case obj2
   ((number complex real) 'rational)
   ((integer) 'integer)
   (else (list 'intersection obj1 obj2))))

                                   ; If OBJ1 is INTEGER
((integer)
 (case obj2
   ((number complex real rational) 'integer)
   (else (list 'intersection obj1 obj2))))

                                   ; If OBJ1 is SSTRING,
                                   ; SYMBOL, or CHAR

((sstring symbol char)
 (list 'intersection obj1 obj2))

                                   ; If OBJ1 is PAIR

((pair)
 (case obj2
   ((slist) 'slist)
   (else (list 'intersection obj1 obj2))))

                                   ; If OBJ1 is SLIST

((slist)
 (case obj2
   ((pair) 'slist)
   (else (list 'intersection obj1 obj2))))
```

```
                                          ; If OBJ1 is PROCEDURE or BOOLEAN

          ((procedure boolean)
           (list 'intersection obj1 obj2))))

;; ————Begin current edits

;; In this section we deal with the case of OBJ1 being a core set and
;; OBJ2 being an explicit set. Basically, we call
;; intersection-core-explicit and be done with it.

        ((explicit? obj2)

                                          ; OBJ1 is CORE, and OBJ2 is
                                          ; EXPLICIT.

         (let ((val2 (set->list obj2)))

                                          ; Call
                                          ; intersection-core-explicit.

           (let ((result (intersection-core-explicit obj1 val2 '())))
               (if (null? result)
                                          ; There were no elements in common.

                   (set )

                                          ; If both are ordered...

                   (if (and (ordered? obj1) (ordered? obj2))

                                          ; Make the result an ordered set.
                       (list->ordered result)

                                          ; One or both are not ordered.
                                          ; Make the result unordered.

                       (list 'union obj1
                             (list->unordered
                              (union-core-explicit
                               obj1 val2 '())))))))))

;; ————End current edits

      ((and (explicit? obj1) (explicit? obj2))
```

```
(letrec
    ((val1 (set->list obj1))
     (val2 (set->list obj2))
     (internal              ; Define an internal recursive
                            ; procedure.
      (lambda (a b intersect)

                            ; A and B are lists containing
                            ; the members of the input
                            ; sets, and INTERSECT is a
                            ; list containing the elements
                            ; of the intersection of the
                            ; two sets. A should be no
                            ; larger than B for optimal
                            ; efficiency.

        (if (equal? a '())  ; Is A empty?
                            ; We're done - return intersection.
            (cons 'set intersect)

                            ; A not empty. Check to see if
                            ; (car A) in B...

            (if (member (car a) b)

                            ; ...it is. Call internal
                            ; with updated INTERSECT list.

                (internal (cdr a) b (cons (car a) intersect))

                            ; Otherwise, call with
                            ; original intersect list.

                (internal (cdr a) b intersect))))))


                            ; We have defined internal.
                            ; Now, apply it to the
                            ; arguments (which we already
                            ; know are both sets).

                            ; If the length of obj1 is
                            ; greater than the length of
                            ; obj2...

  (if (> (length val1) (length val2))
```

```
                                          ; Put the smaller set first.

                 (internal val2 val1 '())
                 (internal val1 val2 '()))))
     (else
      `(intersection ,obj1 ,obj2)))))
    ))
```

difference `difference` is a binary operation yielding the set A - B.

```
(define difference
  (function metaset ((obj1 metaset)
                     (obj2 metaset))
                                          ; Check to see if both
                                          ; arguments are explicit.

    (if (and (explicit? obj1) (explicit? obj2))
        (letrec ((internal                ; Define an internal recursive
                                          ; procedure.
               (lambda (a b diff)

                                          ; A and B are lists containing
                                          ; the members of the input
                                          ; sets, and DIFF is a list
                                          ; containing the elements of A
                                          ; not in B. A should be no
                                          ; larger than B for optimal
                                          ; efficiency.

                 (if (equal? a '())       ; Is A empty?
                                          ; We're done – return intersection.
                     (cons 'set diff)

                                          ; A not empty. Check to see if
                                          ; (car A) in B...

                     (if (member (car a) b)

                                          ; ...it is. Call internal
                                          ; with original DIFF list.

                         (internal (cdr a) b diff)

                                          ; Otherwise, call with
                                          ; updated DIFF list.
```

```
                        (internal (cdr a) b (cons (car a) diff)))))))

                                            ; We have defined internal.
                                            ; Now, apply it to the
                                            ; arguments (which we already
                                            ; know are both sets).


          (internal (cdr obj1) (cdr obj2) '()))

        `(difference ,obj1 ,obj2))))
```

cross The `cross` operator results in the cartesian product of two or more sets. This is the basic method for defining compound data types in Sol. The `cross` operator also allows fields to be named, and has a special syntax for defining fields which are sets of one member (constants) and static fields which are available through the set itself and are shared by all instances created through the constructor.

```
(define cross
  (function
   sset set-list
   (letrec
       ((check-args
         (lambda (arguments)
           (if (null? arguments)
               #t
               (if (set? (car arguments))
                   (check-args (cdr arguments))
                   #f)))))
     (if (check-args set-list)
         (cons 'cross set-list)
         (error "Bad arguments to cross (must be list of sets): " set-list)))))
```

2.2.5 Predicates

set? The `set?` predicate determines whether its argument is a set.

```
(define (set? obj)

                                            ; If OBJ is a fundamental or
                                            ; derived set, return true.

  (or (fundamental? obj) (derived? obj)))
```

fundamental? The `fundamental?` predicated determines whether its argument is a fundamental Sol set.

```
(define (fundamental? obj)
                                        ; If OBJ is a core or explicit
                                        ; set, or a gnu.bytecode.Type
                                        ; instance, return true.
                                        ; Otherwise, false.
  (or (core? obj)
      (explicit? obj)
      (classType? obj)))
```

core? The `core?` predicate determines whether its argument is a core Sol set.

```
(define (core? obj)
                                        ; Check against the various cases.
  (case obj
                                        ; Note: SOL is not included in
                                        ; this list to avoid the
                                        ; difficulty of any other set
                                        ; including SOL.
    ((fundamental core explicit derived
                  number complex real rational integer
                  sstring symbol char
                  sset sordered unordered
                  pair slist
                  procedure
                  boolean) #t)
    (else #f)))
```

explicit? The `explicit?` predicated determines whether its argument is a explicit Sol set.

```
(define (explicit? obj)
  (and
   (and
    (list? obj)                 ; Check to see if OBJ is a non-null list
    (not (null? obj)))
   (or
                                        ; Check to see if first element
    (eq? 'set (car obj))                ; is 'SET or 'ORDERED
    (eq? 'ordered (car obj)))))
```

classType? The `classType?` predicate determines whether its argument is a Java class type.

```
(define (classType? obj)
  (isInstance? <gnu.bytecode.Type> obj))
```

derived? The `derived?` predicated determined wheather its argument is a derived Sol set. This is somewhat tricky because derived sets are defined by operations on other sets.

```
(define (derived? obj)
                                          ; First check to see if obj is a list.
  (if (and (not (null? obj)) (list? obj))

                                          ; Assign the first elemth to
                                          ; OPERATION, the second to A,
                                          ; and the third to B.

      (let ((operation (car obj)))
        (case operation
          ((union intersection difference)

                                          ; Make sure that the list is
                                          ; of length three and that the
                                          ; two operands are sets.

           (and (= (length obj) 3) (set? (cadr obj)) (set? (caddr obj))))

          ((cross)

                                          ; Don't do full checking for
                                          ; the cartesian product. Just
                                          ; make sure the length is at
                                          ; least three.

           (>= (length obj) 3))

          (else #f)))
      #f))
```

ordered? The `ordered?` predicate determines whether or not its argument is an ordered set.

```
(define (ordered? obj)
  (if (and
       (list? obj)
```

```
      (not (null? obj)))
     (if (eq? 'ordered (car obj))
         #t
         #f)
     (case obj
       ((number complex real rational integer
                sstring symbol char) #t)
       (else #f))))
```

isInstance? The `isInstance?` operator is a binary operator which determines if an entity is an instance of a Java class.

```
(define (isInstance? type object)
  ((primitive-virtual-method
    <gnu.bytecode.Type> "isInstance" <boolean> (<Object>))
   type object))
```

in? The `in?` predicate is a binary operator which tests an entity for membership in a set.

```
(define (in? obj class)

                                        ; First, check to see of OBJ
                                        ; is an instance (in the sense
                                        ; of a Java class instance) of
                                        ; CLASS.

  (if (classType? class)
      (isInstance? class obj)
                                        ; Check against
                                        ; core classes. This
                                        ; is essentially a re-write of
                                        ; core? with
                                        ; additional tests.


      (case class

                                        ; Test against the container sets.

        ((sol) #t)                      ; Everything is a member of SOL.
        ((fundamental)
         (fundamental? obj))
        ((core)
         (core? obj))
```

```
((explicit)
 (explicit? obj))
((derived)
 (derived? obj))
                                  ; Test against the number sets.
((number)
 (number? obj))
((complex)
 (complex? obj))
((real)
 (real? obj))
((rational)
 (rational? obj))
((integer)
 (integer? obj))

                                  ; Test against Sstring, Symbol,
                                  ; and Character.

((sstring)
 (string? obj))
((symbol)
 (symbol? obj))
((char)
 (char? obj))

                                  ; Test against SSET, SORDERED,
                                  ; and UNORDERED.

((sset)
 (set? obj))
((sordered)
 (ordered? obj))
((unordered)
 (and (set? obj) (not (ordered? obj))))

                                  ; Test against PAIR and LIST

                                  ; NOTE – in Sol, sets are not
                                  ; lists. However, when
                                  ; embedded in scheme they must
                                  ; be represented that way.
                                  ; Therefor, the scheme list?
                                  ; and pair? operators are not
                                  ; accurate when applied to a
                                  ; Sol set because they will
```

```
                                             ; return true.

((pair)
 (and (pair? obj) (not (set? obj))))
((slist)
 (and (list? obj) (not (set? obj))))

                                             ; Test against procedure.
                                             ; NOTE: this is not entirely
                                             ; accurate, as this test will
                                             ; inaccurately clasify Sol
                                             ; categories as procedures.

((procedure)
 (procedure? obj))

                                             ; Test to see if object is #t
                                             ; or #f.

((boolean)
 (or (eq? obj #t) (eq? obj #f)))

                                             ; At this point, we know that
                                             ; CLASS is not a core
                                             ; set, so check to see if it
                                             ; is an explicit or derived set.

(else
 (cond
  ((explicit? class)
                                             ; It is. Check for membership.
   (member obj (cdr class)))
  ((derived? class)
   (let ((operation (car class))
         (operands (cdr class))
         (operand1 (cadr class))
         (operand2 (caddr class)))

     (cond

                                             ; If the operation is union...

      ((eq? operation 'union)
       (or (in? obj operand1) (in? obj operand2)))

                                             ; Intersection...
```

```
              ((eq? operation 'intersection)

               (and (in? obj operand1) (in? obj operand2)))

                                          ; Difference.

              ((eq? operation 'difference)

               (and (in? obj operand1) (not (in? obj operand2))))

                                          ; The cartesian product
                                          ; operator is different – it
                                          ; may have two or more
                                          ; arguments. We need to
                                          ; define a recursive procedure
                                          ; for checking each element of
                                          ; the list against each
                                          ; argument of CROSS.

              ((eq? operation 'cross)
               (letrec
                   ((recursive-check
                     (lambda (a b)
                       (or (and (null? a) (null? b))
                           (and (not (or (null? a) (null? b)))
                                (in? (car a) (car b))
                                (recursive-check (cdr a) (cdr b)))))))

                 (and (list? obj) (recursive-check obj operands))))

              (else (error "subset?: Bad derived set:" class)))))

                                          ; CLASS is not a Sol
                                          ; set. Return false, since obj
                                          ; can't possibly be a member.
          (else #f))))))
```

subset? This is a predicate taking two sets as arguments. It returns true if $A \subset B, A \neq B$. Subset is important because it provides the mechanism for "typechecking" category inheritance in the new Sol category scheme[2].

```
(define subset?
  (function
```

[2]As of this writing, Fri Jun 12 20:56:27 CDT 1998, the new scheme has not yet been implemented.

```
  boolean ((a sset) (b sset))

                                        ; First, test to see if a and
                                        ; b are the same. This check
                                        ; obviates the necessity of
                                        ; making this comparison for
                                        ; all the other cases, and
                                        ; could be turned off to
                                        ; create a PROPER-SUBSET?
                                        ; operator.

  (if (eq? a b)
                                        ; They are – return true.
      (return #t))

                                        ; Check to see if B is Sol.
                                        ; If it is, return true.

  (if (eq? b sol)
      (return #t))

;; More compilated checking is now required. Begin by creating an
;; internal recursive procedure for testing each member of a list for
;; membership in a set.

  (letrec

                                        ; Define an internal recursive
                                        ; procedure for testing each
                                        ; memeber of a list A for
                                        ; membership in a set THE-SET.
      ((test-contents
        (lambda (whats-left the-set)
          (cond

                                        ; If whats-left is empty, that
                                        ; means every element of B was
                                        ; a member. Return true

           ((null? whats-left)
            #t)

                                        ; If the car of whats-left is in B...

           ((in? (car whats-left) the-set)
            (test-contents (cdr whats-left) the-set))
```

; Otherwise false.

```
(else #f)))))
```

;; Now that the procedure is defined, consider the various cases. The
;; approach used here is to examine the type of sets and make
;; decisions on a case-by-case basis.

```
(cond ((core? a)
```

;; Consider the cases where A is a core set.

```
(if
 (core? b)
```

;; Both A and B are core sets.

; Both sets are core sets, but
; not the same.
; Some core sets are subsets of
; others, so make comparisons
; on case-by-case basis.

```
(case b
  ((sol) (return #t))         ; Every core set is a subset of SOL.

                              ; All core sets are subsets of SSET.
  ((sset)
   (case a
     ((sordered unordered
                fundamental core explicit derived
                number complex real rational integer
                sstring symbol char
                pair slist
                procedure
                boolean) #t)
     (else #f)))
```

; Use the ORDERED? operator to
; decide if a core set is an
; ordered set.

```
  ((sordered) (return (ordered? a)))
```

```
                                 ; The complament (with respect
                                 ; to the core sets) of the
                                 ; ordered sets is the
                                 ; unordered sets.

((unordered) (return (not (ordered? a))))

                                 ; CORE is a subset of fundamental.

((fundamental)
 (case a
   ((core) #t)))

                                 ; Every number set is a
                                 ; subset of NUMBER.
((number)
 (case a
   ((complex real rational integer) (return #t))
   (else (return #f))))

                                 ; Every number set except
                                 ; NUMBER is a subset of
                                 ; COMPLEX.

((complex)
 (case a
   ((real rational integer) (return #t))
   (else (return #f))))

                                 ; RATIONAL and INTEGER
                                 ; are proper subsets of REAL.

((real)
 (case a
   ((rational integer) (return #t))
   (else (return #f))))
((rational)

                                 ; INTEGER is a proper subset
                                 ; of RATIONAL.

 (case a
   ((integer) (return #t))
   (else (return #f))))
```

```
                                ; LIST is a proper subset of PAIR.
   ((pair)
    (case a
      ((slist) (return #t))
      (else (return #f))))

                                ; INTEGER, SSTRING, SYMBOL,
                                ; CHAR, PROCEDURE, AND BOOLEAN
                                ; have no proper subsets among
                                ; the core classes.

   ((explicit derived
              integer
              sstring symbol char
              unordered
              slist
              procedure
              boolean)
    (return #f)))


                                ; The only way A can be a
                                ; subset of B if A is core is
                                ; if B is also core.
                                ; Therefor, return false.

  (return #f)))
```

;; Done with the first cond clause.
;; A isn't a core set. Check to see if it is a fundamental set.

```
 ((explicit? a)

                                ; A is an explicit
                                ; user-defined set.

  ((fundamental? b)

                                ; A is explicit user-defined
                                ; and B is a fundamental set,
                                ; meaning that it could be
                                ; core or explicit.


                                ; Make a into a list and
                                ; assign the results to
```

```
                              ; CONTENTS.

      (let ((contents (set->list a)))

                              ; We have to explicitly check
                              ; for the empty set condition,
                              ; because TEST-CONTENTS won't
                              ; detect it.

       (if (null? contents)
           (return (in? a b))


                              ; Return the results of
                              ; calling TEST-CONTENTS on the
                              ; contents of A and B.

           (return (test-contents contents b)))))

;; A is explicit - is B derived?

     ((derived? b)

                              ; A is explicit, user-defined
                              ; and B is derived. Most of
                              ; the cases will involve
                              ; recursively calling SUBSET?
                              ; to get the answer.

                              ; Assign some conveneint variables.

      (letrec ((operation (car b))
               (operand1 (cadr b))
               (operand2 (caddr b)))

                              ; Check for different
                              ; operation types.

       (case operation

                              ; If the operation is union...

        ((union)
         (return
          (or (subset? a operand1) (subset? a operand2))))
```

```
                                      ; Intersection...

((intersection)

                                      ; *** This isn't always true.
                                      ; The intersection could be
                                      ; the empty set, which may not
                                      ; be a member of B. Likewise,
                                      ; neither set may be a subset
                                      ; of B, but the intersection
                                      ; certainly could be. I will
                                      ; have to fix this in the
                                      ; future.

 (return
  (and (subset? a operand1) (subset? a operand2))))

                                      ; Difference.

((difference)

                                      ; If operand1 is a subset of
                                      ; operand2, the difference is
                                      ; the empty set. Check for
                                      ; this condition, and return
                                      ; true in this case only if
                                      ; the empty set is a subset of
                                      ; operand 2.

 (if (subset? operand1 operand2)
     (return (subset? a (set)))

                                      ; Operand1 is not a subset of
                                      ; operand2. Therefor,
                                      ; operand1 - operand2 will
                                      ; leave some elements of
                                      ; operand 1.

     (return (subset? a operand1))))

                                      ; Check for a cartesian product.

((cross)

                                      ; Check every member of A for
                                      ; membership in B. Return
```

```
                                       ; true if all are members.

          (let ((contents (set->list a)))

                                       ; We have to explicitly check
                                       ; for the empty set condition,
                                       ; because TEST-CONTENTS won't
                                       ; detect it.

            (if (null? contents)
                (return (in? a b))

                                       ; Return the results of
                                       ; calling TEST-CONTENTS on the
                                       ; contents of A and B.

                (return (test-contents contents b)))))

                                       ; Error - operation must be
                                       ; union, intersection or
                                       ; difference.

         (else (error "subset?: Bad derived set:" b))))))


;; Is A a derived set?

      ((derived? a)
                                       ; A is a derived set.
                                       ; Recursively call subset? in
                                       ; the appropriate
                                       ; combinations.

       (letrec ((operation (car a))
                (operand1 (cadr a))
                (operand2 (caddr a)))

          (case operation

                                       ; If the operation is union...

           ((union)
            (return
             (and (subset? operand1 b) (subset? operand2 b))))

                                       ; Intersection...
```

```
((intersection)

                                ; *** This isn't always true.
                                ; The intersection could be
                                ; the empty set, which may not
                                ; be a member of B. Likewise,
                                ; neither set may be a subset
                                ; of B, but the intersection
                                ; certainly could be. I will
                                ; have to fix this in the
                                ; future.

 (return
  (and (subset? operand1 b) (subset? operand2 b))))

                                ; Difference.

((eq? operation 'difference)

                                ; If operand1 is a subset of
                                ; operand2, the difference is
                                ; the empty set. Check for
                                ; this condition, and return
                                ; true in this case only if
                                ; the empty set is a subset of
                                ; operand 2.

 (if (subset? operand1 operand2)
     (return (subset? (set) b))

                                ; Operand1 is not a subset of
                                ; operand2. Therefor,
                                ; operand1 - operand2 will
                                ; leave some elements of
                                ; operand 1.

     (return (subset? operand1 b))))

                                ; Error – operation must be
                                ; union, intersection or
                                ; difference.

(else (error "subset?: Bad derived set:" a)))))

                                ; A is not a set, so it cannot
```

; possibly be a subset of B.

```
          (else (return #f)))))
)
```

2.2.6 Standard procedures relating to Sol sets

Conversion procedures

set→list The `set→list` function converts a set to a list.

```
(define set->list
  (function slist
            ((a sset))
            (if (ordered? a)
                (vector->list (cadr a))
                (cdr a))))
```

ordered→vector The `ordered→vector` function converts an ordred set to a Scheme vector.

```
(define ordered->vector
  (function sol
            ((a sordered))
            (cadr a)))
```

The `ordered-ref` function extracts an indexed element of an ordered set.

```
(define ordered-ref
  (function sol
            ((a sordered)
             (index integer))
            (vector-ref (cadr a) index)))
```

list→unordered The `list→unorderd` function converts a list to an unordered set.

```
(define list->unordered
  (function sset
            ((a slist))
            (cons 'set a)))
```

list→ordered The `list→ordered` function converts a list to an ordered set.

```
(define list->ordered
  (function sordered
            ((a slist))
            (list 'ordered (list->vector a))))
```

Ordered set operations

element The `element` function accesses an element of an ordered set by index.

```
(define element
  (function sol
          ((a sordered)
           (i integer))
          (if (fundamental? a)
              (vector-ref (cadr a) i)
              #f)))
```

Iteration

The `parallel` function takes as its arguments a set, an operator, and possibly additional arguments. It applies the operator to the set in parallel, and returns the empty set. `Parallel` may be called for its side effects, or as a means of manipulating the set elements. This implementation of `parallel` is actually sequential, but it is an error to use `parallel` in a way which relies on a particulat order of evaluation.

```
(define parallel
  (function sset
          ((a sset) (b procedure) . c )
          (letrec
              ((internal-ordered
                (lambda (vector index length)
                  (if (< index length)
                      (begin (apply
                              b
                              (cons
                               a
                               (cons (vector-ref vector index)
                                     (cons index c))))
                             (internal-ordered
                              vector (+ 1 index) length))
                      '())))
               (internal-not
                (lambda (elements operation)
                  (if (null? elements)
                      '()
                      (begin (apply operation
                                    (cons a (cons (car elements)
                                                  (cons -1 c))))
                             (internal-not (cdr elements) operation))))))
            (if (ordered? a)
                (internal-ordered
```

```
                (cadr a) 0 (vector-length (cadr a)))
               (internal-not (cdr a) b))
           '(set))))
```

operate The operate function takes as its arguments a set, an operator, and possibly additional arguments. It returns a new set which is the result of the parallel evaluation of every member of the original set using the operator.

```
(define operate
  (function sset
          ((a sset) (b procedure) . c )
          (letrec ((internal-ordered
                    (lambda (vector index length)
                      (if (< index length)
                          (union (apply
                                  b
                                  (cons a
                                        (cons (vector-ref vector index)
                                              (cons index c))))
                                 (internal-ordered
                                  vector (+ 1 index) length))
                          (ordered ))))
                   (internal-not
                    (lambda (elements operation)
                      (if (null? elements)
                          (set)
                          (union (apply operation
                                        (cons a
                                              (cons (car elements)
                                                    (cons -1 c))))
                                 (internal-not (cdr elements) operation))))))
            (if (ordered? a)
                (internal-ordered
                 (cadr a) 0 (vector-length (cadr a)))
                (internal-not (cdr a) b)))))
```

The synchronize statement takes a thunk (a procedure of no arguments) and evaluates the thunk in a synchronized (thread-safe) procedure.

```
(define
  (synchronize thunk)
  ((primitive-static-method
    <sol.java.Synchronized> "apply0" <Object> (<gnu.mapping.Procedure0>))
   thunk))

(define
  (get-current-environment)
```

```
  ((primitive-static-method
    <gnu.mapping.Environment> "getCurrent" <gnu.mapping.Environment> ())))

(defmacro include (sourcefile)
  (if (in? sourcefile sstring)
      `(load ,(string-append "/mas/disks/acg2/u2/rich/projects/sol"  "/" sourcefile))
      `(error "include: Argument not a string."
              (list in? sstring ,sourcefile) '-> #f)))
```

2.3 Libraries

This section documents the libraries which are part of the Sol langauge system. Not all libraries are available under all implementations.

2.3.1 The SolCanvas Graphics System

The SolCanvas graphics system is implemented as a Sol category. The member-set of the category is the return-type of many member functions.

```
(define SolGraphics
  (category
   ()                                    ; No parent category

                                         ; Set of objects the
                                         ; SolGraphics functions will
                                         ; manipulate. Starts out
                                         ; empty. As new objects are
                                         ; created, they are added to
                                         ; the set.

   (set "SolCanvas: Error")

   ()                                    ; No set operators

                                         ; Define the list of member methods.

;; The newFrame method produces a new AWT frame with the specified
;; title, width, and height, and adds it to the member-set. This
;; method is primarily for creating Sol applications rather than
;; applets (which will inherit a frame from the SolApplet java class).

   ((newFrame
     (function
      member-set ((title sstring)
                  (width integer)
                  (height integer))
      (let ((frame
             ((primitive-constructor
               <java.awt.Frame> (<java.lang.String>)) title)))
        (set-set! (union member-set (set frame)))
        (setFrameSize frame width height)
        (return frame))))

;; The setFrameSize method sets he size of a frame.

    (setFrameSize
     (function
      sol ((frame member-set)
           (width integer)
           (height integer))
      ((primitive-virtual-method
```

```
     <java.awt.Frame> "setSize" <void> (<int> <int>)) frame width height)))
```

;; The showFrame method displays a frame previously created by
;; newFrame.

```
(showFrame
 (function
  sol ((frame sol))
  ((primitive-virtual-method <java.awt.Frame> "show" <void> ()) frame)))
```

;; The getCanvas method gets a drawing canvas from a SolApplet
;; subclass instance. This is what Sol applets use to get a drawing
;; context.

```
(getCanvas
 (function
  member-set ()
  (let ((canvas2
         ((primitive-get-static
           <sol.java.SolApplet> "solCanvas" <sol.java.SolCanvas>))))
    (set-set! (union member-set (set canvas2)))
    (return canvas2))))
```

;; The newSolCanvas method creates a new SolCanvas with the specified
;; update delay time (in miliseconds), and adds it to the member-set.

```
(newSolCanvas
 (function
  member-set ((delay integer))
  (let ((canvas
         ((primitive-constructor <sol.java.SolCanvas> (<long>)) delay)))
    (set-set! (union member-set (set canvas)))
    (return canvas))))
```

;; The updatePosition method changes the x,y location of a drawable
;; and updates the bounding box assuming the size has not changed.

```
(updatePosition
 (function
  sol ((drawable sol)
       (x integer)
       (y integer))
  ((primitive-virtual-method
    <sol.java.Drawable> "updatePosition" <void> (<int> <int>))
   drawable x y)))
```

;; The newCircleDrawable method creates a new CircleDrawable with the
;; specified center, radius, color, and fill-flag.

```
(newCircleDrawable
 (function
  sol ((x integer)
       (y integer)
       (radius integer)
       (color sol)
       (drawMe boolean))
  ((primitive-constructor
    <sol.java.CircleDrawable>
    (<int> <int> <int> <java.awt.Color> <boolean>))
   x y radius color drawMe)))
```

;; The newLineDrawable method creates a new LineDrawable with the
;; specified coordinates and color.

```
(newLineDrawable
 (function
  sol ((x1 integer)
       (y1 integer)
       (x2 integer)
       (y2 integer)
       (color sol))
  ((primitive-constructor
    <sol.java.LineDrawable>
    (<int> <int> <int> <int> <java.awt.Color>)) x1 y1 x2 y2 color)))
```

;; The newPolyDrawable method creates a new PolyDrawable with the
;; specified coordinates and color.

```
(newPolyDrawable
 (function
  sol ((x integer)
       (y integer)
       (color sol)
       (drawme boolean))
  ((primitive-constructor
    <sol.java.PolyDrawable>
    (<int> <int> <java.awt.Color> <boolean>)) x y color drawme)))
```

;; The addPolyPoint method adds a point to a PolyDrawable.

```
(addPolyPoint
```

```
    (function
     sol ((poly sol)
          (x integer)
          (y integer))
     ((primitive-virtual-method
       <sol.java.PolyDrawable> "addPoint" <boolean> (<int> <int>))
      poly x y)))
```

;; The addDrawable method adds a drawable to a SolCanvas.

```
   (addDrawable
    (function
     sol ((canvas member-set)
          (drawable sol))
     ((primitive-virtual-method
       <sol.java.SolCanvas> "addDrawable" <void> (<sol.java.Drawable>))
      canvas drawable)))
```

;; The removeAllDrawables method removes al drawables from a SolCanvas.

```
   (removeAllDrawables
    (function
     sol ((canvas member-set))
     ((primitive-virtual-method
       <sol.java.SolCanvas> "removeAllDrawables" <void> ()) canvas)))
```

;; The paint method paints a SolCanvas

```
   (paint
    (function
     sol ((canvas member-set))
     ((primitive-virtual-method
       <sol.java.SolCanvas> "paint" <void> ()) canvas)))
```

;; The add method adds a component (like a SolCanvas) to a container
;; (like a Frame)

```
   (add
    (function
     sol ((container member-set)
          (location sstring)
          (component member-set))
     ((primitive-virtual-method
       <java.awt.Container>
       "add" <java.awt.Component> (<String>
                                   <java.awt.Component>))
```

```
    container location component)))
```

;; The nameColor method gets an AWT color by name

```
(nameColor
 (function
  sol ((name sstring))
  ((primitive-get-static <java.awt.Color> name <java.awt.Color>))))
```

;; The rgbColor method gets an AWT color by RGB values.

```
(rgbColor
 (function
  sol ((red real)
       (green real)
       (blue real))
  ((primitive-constructor
    <java.awt.Color> (<float> <float> <float>)) red green blue)))
```

;; The setBackground method sets the background color of a component.

```
(setBackground
 (function
  sol ((component member-set)
       (color sol))
  ((primitive-virtual-method
    <java.awt.Component>
    "setBackground" <void> (<java.awt.Color>)) component color)))
```

;; The setForeground method sets the foreground color of a component.

```
(setForeground
 (function
  sol ((component member-set)
       (color sol))
  ((primitive-virtual-method
    <java.awt.Component>
    "setForeground" <void> (<java.awt.Color>)) component color)))

)))
```

2.4 Postscript

Inform the user that Sol is ready.

```
(newline) (display "Sol $Revision: 2.22 $ WALRUS") (newline)
```