Chapter 3. Extending Scheme

Topic: *Scheme Interface, *Extending ACIS

The true versatility of Scheme arises from the ability to extend it. Application developers can quickly tailor the Scheme language to their needs by creating language *extensions* in C++, and then by writing interpreted Scheme procedures that use those extensions.

This chapter describes how to write Scheme extensions (which are Scheme primitive expressions) to enhance the functionality provided in the Scheme ACIS Interface Driver Extension (Scheme AIDE). This chapter assumes that the user is already familiar with the Scheme language.

Scheme AIDE is based on the Elk Extension Language, with additional data types to support ACIS modeling, additional data access, inquiry, conversion, and extension procedures. Scheme AIDE allows ACIS applications to be developed using the Scheme language and the ACIS Scheme extensions. Scheme AIDE interprets Scheme procedures, calls the appropriate C++ handler function, and converts data between C++ and Scheme data types.

Scheme AIDE is a demonstration application. Refer to the Scheme Support Component for information about running this application and using Scheme AIDE as an example Scheme based application.

Whenever new Scheme extensions are created, the Scheme AIDE application must be rebuilt to make the extensions available ("known" to the Scheme Interpreter).

Scheme Extensions

Topic: *Extending ACIS, *Scheme Interface

Scheme extensions are programs or functions that are compiled and linked into Scheme AIDE. Thereafter, they are available for use at Scheme application's command line or in Scheme procedures.

Scheme extensions are categorized as:

object–type:action	These procedures perform some kind of action on the type of object described by object–type. An example of this type of procedure is (entity:set–color my_ent color), which performs the action "set color" on an object of type "entity". The first argument for this type of procedure is the object on which the specified action is performed.

object–type:output	These procedures perform a query operation on an object described by object–type. An example of this kind of procedure is (position:x my_pos), which returns the *x*–component of a user defined position, my_pos. These procedures are considered as a special case of object–type:action, where the action is to query some value, but the get portion of the action is eliminated to make the name shorter.
object–type:outputs	This form is similar to the object–type:output form, but it does a query when more that one value can be returned. The values are always returned as a list. In cases in which the object being queried has no values that satisfy the query, an empty list is returned. An example of this kind of procedure is (entity:faces my_ent), which returns a list of all of the faces associated with the given entity, my_ent.

Extensions to Scheme

Topic: *Extending ACIS, *Scheme Interface

ACIS contains extensions to the native Scheme language. These extensions fall into three categories:

- *Data types and methods.*
- *Scheme primitive procedures* (new commands), which are composed of a C++ handler function and a binding to some Scheme character string (command string) that is passed into the Scheme Interpreter to invoke the handler function. These are referred to as ACIS *Scheme extensions*.
- *Initialization functions in C++*, which set startup conditions when Scheme AIDE is initialized.

The application developer can create new extensions in each of these categories, specific to the needs of the application. These extensions, defined in C++ and bound to the interpreter, are known to the interpreter at initialization time and are used immediately. (Scheme procedures, however, differ, because they are loaded or defined at run time, and do not constitute a Scheme language extension.)

Application developers can extend the Scheme interface by defining new Scheme extension commands specific to their application. These commands are then known to the Scheme Interpreter when it is initialized, just like any native Scheme primitive (such as define, let, etc.). Any number of new Scheme extensions can be created.

An extension consists of a handler function, and a call to the SCM_PROC macro that associates the handler with some string to be entered by the user.

Defining New Scheme Extensions

Topic: *Scheme Interface

The following steps create a new Scheme extension:

1. Assign a file name for the **C++** handler function.

   All file names are eight characters or less with a three character extension to comply with DOS conventions (even on UNIX platforms). A file that implements Scheme procedures in C++ has a name like foo_scm.cxx where foo is derived from the name of the procedure. For example, handler procedures for creating solids are defined in the sld_scm.cxx file and declared in the sld_scm.hxx file. Underlying support functions are placed in a the sld_utl.cxx file and declared in the sld_utl.hxx file.

2. Create a standard comment header block that describes the command and its arguments.

3. Create a procedure name.

   The name of the C++ handler function that implements a Scheme procedure derives from the name of the Scheme procedure. The handler name starts with P_ to indicate that it is defining a Scheme procedure. The remaining parts of the name are the components of the Scheme procedure name separated with underscores, and with the first letter of each component capitalized. If the Scheme procedure ends in a question mark, end the C++ procedure's name with the letter p (this is derived from common LISP usage).

4. Define arguments and return type.

   Every Scheme procedure takes Scheme objects as arguments and returns a Scheme object. If there are a fixed number of arguments, then each object is a Scheme object. If there are a variable number of arguments, then the first argument is an int that gives the number of arguments, and the second argument is an array of ScmObjects (similar to ARGC and ARGV).

5. Call the ENTER_FUNCTION macro.

   The first statement of every procedure is an ENTER_FUNCTION macro call. ENTER_FUNCTION is defined in the trace.hxx file and is used as a debugging aid to trace program execution. It takes one string argument, which gives the name of the procedure.

6. Access data.

   The main task of the function is to convert the arguments into the appropriate C++ data types, and then do something with that data. In most cases, the handler function does little actual computation itself. Instead, if there is any appreciable amount of computation to be done, write another procedure to do it, and put it into a separate source file.

7. Convert data to C++.

   If an argument can only be a single data type, then use the Scheme get procedures to convert it to the C++ type. For example, if the first argument in a procedure is a position, use get_Scm_Position to convert it from a ScmObject to an ACIS position. The get procedure signals an error if a ScmObject of the wrong type is passed in, so there is no need to check the data type explicitly.

   If an argument is more than one data type (for example, a number or a position), then use the is_scm_xxx routines to determine the data type. If it is not a valid type, call Wrong_Type_Combination to signal a data type error.

8. Create a new ENTITY.

   The start_entity_creation and end_entity_creation procedures update the view based on changes to the part. The entity creation is performed in an API routine or in a routine that looks like an API routine. The API functions add an entity to the part. The part controls creation of bulletin boards, delta states, and roll back.

9. Return the result.

   The make_Scm_Entity call converts an ACIS entity into a Scheme entity. All Scheme procedures must return a Scheme object. The ScmObject is created with a call to make_Scm_xxx for the type of object created. If the return is the equivalent of void in C++, then use the make_unspecified call to create the returned ScmObject.

10. Bind a user–typed command to the handler procedure by calling SCM_PROC macro.

    The SCM_PROC macro provides dynamic run–time binding between Scheme and C++ to allow Scheme AIDE to call the appropriate C++ handler functions. Each new extension to Scheme AIDE must be "registered" using this macro.

    SCM_PROC creates a static instance of a user_scm_proc object. The constructor for user_scm_proc links it into a list of all instances of that class. At startup time, an initialization procedure is called that invokes a register method for all of the objects in the list that registers the procedures with the Scheme Interpreter.

    A Scheme procedure has a variable number of arguments. This is indicated by giving different numbers for the maximum and minimum arguments to SCM_PROC. The Scheme Interpreter checks that the number of arguments supplied by the user is within the given range, so it is not necessary to implement this check with the C++ handler procedure.

    The macro's format is:

    ```
    SCM_PROC(n1, n2, name, proc)
    ```

where:

n1
is the minimum number of parameters to the handler function.

n2
is the maximum number of parameters to the handler function.

names
is the command string that invokes the handler function.

proc
is the C++ handler function to be invoked.

Solid Sphere Example

Topic: *Scheme Interface

Example 3-1 is in the sld_scm.cxx file. It creates a solid sphere when the user enters the command (solid:block <center point> <radius>). All Scheme extensions, such as this one, use the standard naming, header, and other conventions listed in the preceding section.

```
// Purpose–––Create a sphere given the center and the radius.
//
// syntax–––
//      syn: (solid:sphere center–position radius)
//      syn#/   center–position = A 3D position
//      syn#/   radius          = A real
//      syn#/   returns         = An entity

ScmObject P_Solid_Sphere(ScmObject p1, ScmObject r)
{
    ENTER_FUNCTION(”P_Solid_Sphere”);

    // Get the center and the radius.
    position center = get_Scm_Position(p1);
    double radius = get_Scm_Real(r);

    // Make the sphere.
    BODY *sphere = NULL;
    start_entity_creation();
    outcome result = api_solid_sphere(center, radius, sphere);
    end_entity_creation(sphere, result);

    return make_Scm_Entity(sphere);
}

SCM_PROC(2, 2, ”solid:sphere”, P_Solid_Sphere);
```

Example 3-1. Creating a Solid Sphere

The following items are specified in Example 3-1:

Handler procedure	is of the type Scheme object, is named P_Solid_Sphere, and takes exactly two parameters: a center and a radius.
ENTER_FUNCTION	is a diagnostic hook macro commonly used by *Spatial* for stack tracing and other debugging. During normal operations, it does nothing. If the code is compiled with the DEBUG_TRACE flag defined, ENTER_FUNCTION prints the name of the function each time the function is entered.
Procedure body	extracts C++ objects from the Scheme objects passed as parameters, and calls ACIS functions to create a sphere.
Procedure returns	is a Scheme object of the type entity, which is a sphere.
SCM_PROC	registers this extension to the Scheme Interpreter. The SCM_PROC line shown here indicates that this procedure can have no less than two parameters, no more than two parameters, is invoked with the solid:sphere string, and calls the P_Solid_Sphere function.

To create a sphere of radius 10 centered about the origin, the user inputs (by typing or giving Scheme AIDE a procedure file) the following:

```
(solid:sphere (position 0 0 0) 10)
```

The call supplies two parameters. The first parameter is a position consisting of three values, and the second parameter is a radius.

Scheme Data Types and Methods

Topic: *Scheme Interface

ACIS defines a set of data types and methods for the native Scheme language. Data types and methods do the following:

- Determine if a Scheme object (ScmObject) is an object of a particular type.
- Convert a Scheme object into the corresponding C++ object.
- Convert a C++ object into the corresponding Scheme object.
- Evaluate Scheme expressions from C++.
- Enable application developers to define their own new data types and Scheme procedures.

For descriptions of the data types that ACIS provides, refer to their reference templates.

Scheme AIDE provides data types specifically for use with ACIS Scheme extensions, in addition to those that are native to Scheme. These data types are described in online help. Some data types are specific to ACIS, such as entity, pick–event, and gvector. Other data types are native to Scheme, such as boolean, integer, and real. Native data types are documented in online help only if they are needed to describe the ACIS data types.

The reference template for Scheme extensions includes an *Arg Types* field that specifies the data type of each object passed into the extension and a *Returns* field that specifies the data type of the object returned by the extension. In some cases, the return of an extension is unspecified, and the *Returns* field contains the word unspecified. This is not an actual data type, only an indicator of the lack of a specified data type returned by the extension.

At their highest level, all Scheme object data types are scheme–objects. When an ACIS Scheme extension is invoked, a C++ handler procedure generally converts the scheme–object arguments to their corresponding C++ types, performs ACIS or other operations, then converts the result back to a scheme–object, which is returned by the extension.

At their next highest level, many types of arguments are entitys. entity objects are saved and restored as part of the model and are referred to as *persistent*. Non–entity objects are not saved and restored as part of the model (they are lost when Scheme AIDE is terminated) and are referred to as *transient.*

At more specific levels, many types of arguments are really *subtypes* of entity. For example, body, lump, shell, face, coedge, edge, vertex, and wire are all topological subtypes of entity.

Figure 3-1 graphically depicts the data type relationships. An item in the tree is read from right to left. For example, a circular–edge is a curve–edge that is an edge that is an entity that is a scheme–object. Individual derivations are shown in each data type's description.

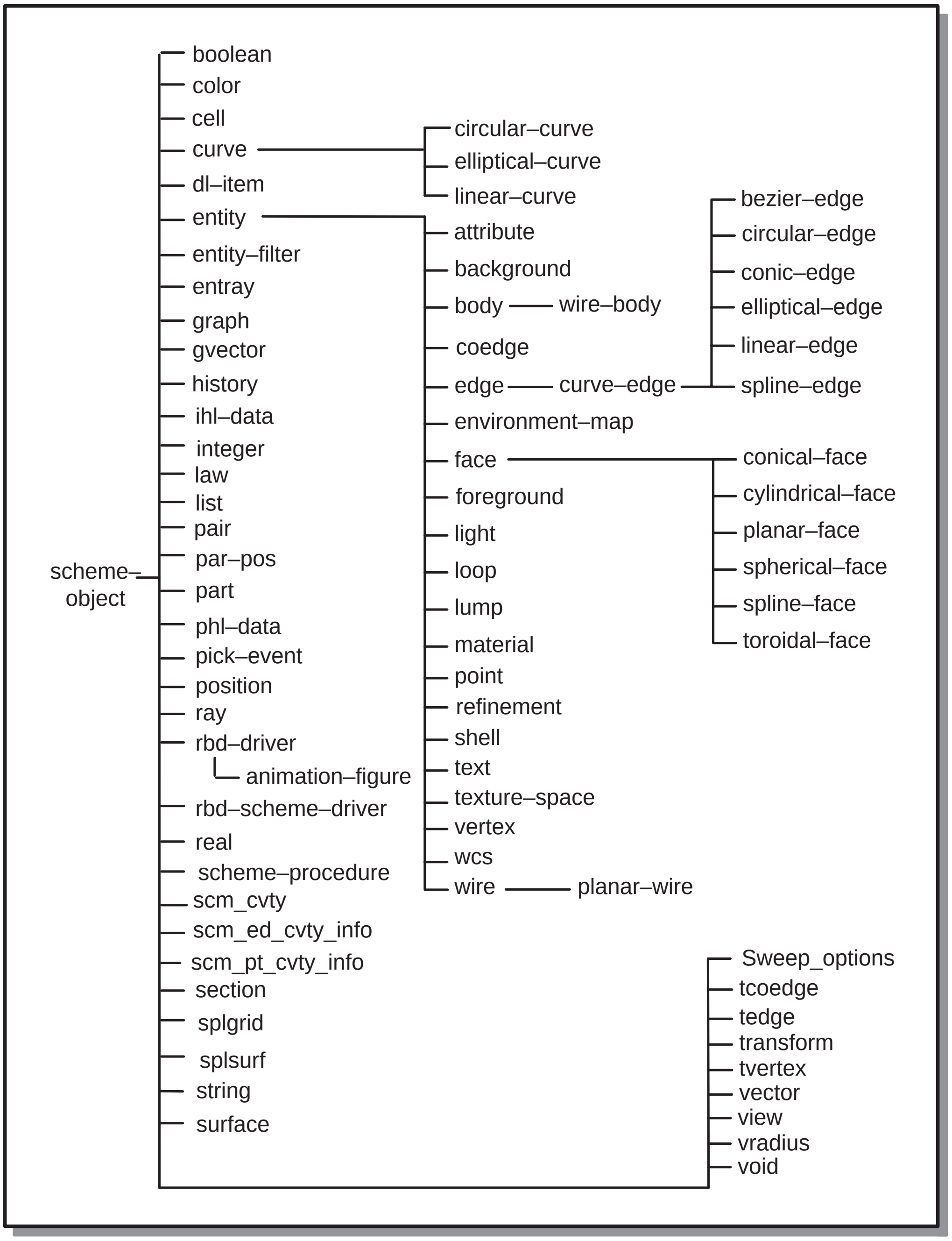

Figure 3-1. Scheme Data Type Relationships

Data types immediately derived from scheme–object that have unique external representations are *true data types* implemented in Scheme or in extensions. Other data types without distinct external representations are not true data types, but *subclassifications* used only as a convenience for the purposes of discussion.

Entity Objects

Topic: *Scheme Interface

Many examples produce entity objects, and then use them. For example:

```
(solid:block (position 0 0 0) (position 10 10 10))
;; #[entity 2 1]
```

The solid:block extension creates an entity. Scheme AIDE maintains the external representation of this entity as #[entity 1 1], where 1 is the entity ID, and 0 is the part ID. Subsequent operations that require this entity as input, such as entity:faces, would require the external representation.

```
(entity:faces (entity 1))
;; (#[entity 2 1]#[entity 3 1]#[entity 4 1]
;; #[entity 5 1]#[entity 6 1]#[entity 7 1])
```

Note *Do not confuse the external representation of an entity, as in* #[entity 2 1], *with the Scheme extension called* entity. *The external representation is simply a diagnostic output showing what the extension returned. The* entity *extension accepts an ID number and returns the object.*

Scheme AIDE maintains its own entity numbering for each session. The entity numbers are not always freed up even after the entity is deleted. If an entity is passed into another operation, there are no guarantee that the entity numbers be the same. The way around this is to use define statements and your own variable names. The variable names are then passed into subsequent operations. In this manner, the same set of operations can be performed over and over within the same Scheme AIDE session without having to worry about specific entity numbers.

```
(define my_block (solid:block
    (position 0 0 0) (position 10 10 10)))
;; my_block
; my_block => #[entity 1 1]
(entity:faces my_block)
;; (#[entity 2 1]#[entity 3 1]#[entity 4 1]
;; #[entity 5 1]#[entity 6 1]#[entity 7 1])
```

Lists

Topic: *Scheme Interface

Scheme extensions often take lists of a particular data type as arguments, or return a list of some data type. The notation for a list of arbitrary length of a particular data type is:

```
(<data type> ...)
```

Data Access Procedures

Topic: *Scheme Interface

Procedures are provided to convert Scheme data types and the corresponding C++ data types. There are procedures for both the standard Scheme types and the types that are added for ACIS. The following procedures exist for each data type:

- Determining if a Scheme object is an object of that type
- Converting a Scheme object into the corresponding C++ object
- Converting a C++ object into the corresponding Scheme object

Defining New Scheme Data Types

Topic: *Scheme Interface

Application developers can create new application–specific data types, and methods that enable those data types to be used by both Scheme programs and C++ functions.

Methods include the following:

Constructor . . . . creates an instance of the type (not required).

Destructor . . . . . deletes an instance of the type (not required).

init . . . . initializes the type. It calls Define_Type to actually define the type to Scheme. It is invoked using the SCM_INIT macro.

Equal . . . . compares two instances of the type to determine if they are the same. Call the debug macro, ENTER_FUNCTION(method_name), at the beginning of the function. This method is supplied to Define_Type for the interpreter's use.

Print . . . . prints an instance of the type. Call the debug macro, ENTER_FUNCTION(method_name), at the beginning of the function. This method is supplied to Define_Type for the interpreter's use.

Is . . . . determines if a Scheme object is of this particular type. Call the debug macro, ENTER_FUNCTION(method_name), at the beginning of the function.

Get . . . . creates a *C++* object of this type of Scheme object. Call the debug macro, ENTER_FUNCTION(method_name), at the beginning of the function.

Make . . . . creates a Scheme object of this type from a C++ object. Call the debug macro, ENTER_FUNCTION(method_name), at the beginning of the function.

The following sections analyze the GC_BndCrv class and its methods as an example of how to derive a new data type. GC is an acronym for garbage collection, which is the process of recovering and reusing memory allocated by the Scheme Interpreter. The interpreter never destroys a Scheme object unless it can prove that the object is not used again. It does, however, recycle memory efficiently to prevent excessive memory allocation.

Class Definition

Topic: *Scheme Interface

Developers create new classes either by defining their own classes or by deriving child classes from existing classes.

The following three blocks of code provide examples of creating classes. The code in Example 3-2 derives the GC_BndCrv bounded curve class from its parent, the GC_Object class.

```
// Define a GC_Object class for tracking bounded_curves.

class GC_BndCrv : public GC_Object
{
public:
bounded_curve* The_Bounded_Curve;
GC_BndCrv(ScmObject, bounded_curve*);
virtual ~GC_BndCrv();
};
```

Example 3-2. Deriving a Class from its Parent

The code in Example 3-3 defines the underlying data structure. All Scheme object structures must include the ScmObject tag (for garbage collection) as their first element.

```
struct S_BndCrv {
    ScmObject tag;
    GC_BndCrv* The_GC_BndCrv;
};

#define BNDCRV_PTR(obj) (
((struct S_BndCrv*)POINTER(obj))->
    The_GC_BndCrv->The_Bounded_Curve)
```

Example 3-3. Defining the Underlying Structure

The code in Example 3-4 defines type # assigned by Define_Type. This block checks the object type.

```
// T_BndCrv data type
```

```
static int T_BndCrv;
```

Example 3-4. Defining the Type Number

Constructor Method

Topic: *Scheme Interface

The *constructor* method creates a new instance of this class. The name of the constructor is the same as the name of the class. Example 3-5 constructs a new bounded_curve object.

```
// Purpose---Create a GC_BndCrv object.

GC_BndCrv::GC_BndCrv(ScmObject obj, bounded_curve* bcrv):
GC_Object(obj)
{
    The_Bounded_Curve = bcrv;
}
```

Example 3-5. Creating a New Instance of a Class

Destructor Method

Topic: *Scheme Interface

The *destructor* method destroys an existing instance of this class. The name of the destructor is the same as the name of the class, prefixed by a ~ (not) symbol. Example 3-6 destroys a bounded_curve object.

```
// Purpose---Delete a GC_BndCrv.

GC_BndCrv::~GC_BndCrv()
{
    fprintf(stderr,"Delete bounded_curve %x\n", this);
    delete The_Bounded_Curve;
}
```

Example 3-6. Destroying an Instance of a Class

Initialization Method

Topic: *Scheme Interface

The *initialization* method initializes an object type and defines it to the Scheme Interpreter using the SCM_INIT macro. Initialization methods are named init, followed by an underscore and the type name. Example 3-7 shows the init method init_S_BndCrv.

```
// Purpose---Define S_BndCrv type.

void init_S_BndCrv()
{
    ENTER_FUNCTION("init_S_BndCrv");
    T_BndCrv = Define_Type(0,
        ”curve”,
        (int (*)(size_t))NOFUNC,
        sizeof(S_BndCrv),
        BndCrv_Equal,
        BndCrv_Equal,
        BndCrv_Print,
        (void (*)(ScmObject*, void (*)(ScmObject*)))NOFUNC
        );
}
// Bind procedure to the Scheme Interpreter so that it is called.
SCM_INIT(init_S_BndCrv);
```

Example 3-7. Initializing an Object Type

The following items are specified in Example 3-7:

T_BndCrv = Define_Type	is the requested type number. Always use 0 to request a made–up type number.
“curve”	is the type name for error messages.
(int (*)(size_t))NOFUNC	is the site function for variable site objects
sizeof(S_BndCrv)	is the size for fixed–size objects.
BndCrv_Equal	is the equivalent function for (equ?).
BndCrv_Equal	equal functions for (equal?).
BndCrv_Print	is the print function.
(void (*)(ScmObject*, void	is the visit function, which is needed only if the object contains other Scheme objects.

Equal Method

Topic: *Scheme Interface

The *equal* method compares two Scheme objects (instances) of this class to determine if they are equivalent. This is useful in cases where there might be several pointers all pointing to the same instance. Compare methods are named for the class, followed by an underscore and the word Equal. Example 3-8 compares two Scheme bounded_curve objects to determine if they are the same.

```
// Purpose–––Test if 2 S_BndCrv objects are equivalent.
```

```
int BndCrv_Equal(ScmObject c1, ScmObject c2)
{
    ENTER_FUNCTION(”BndCrv_Equal”);
    return BNDCRV_PTR(c1) == BNDCRV_PTR(c2);
}
```

Example 3-8.   Comparing Two Scheme Objects

Print Method

Topic: *Scheme Interface

The *print* method outputs the object to a port. Print methods are named for the class, followed by an underscore and the word Print. Example 3-9 prints a Scheme bounded_curve object.

```
// Purpose–––Print a S_BndCrv on an output port.

void BndCrv_Print(ScmObject c, ScmObject port, int, int, int)
{
    ENTER_FUNCTION(”BndCrv_Print”);

    bounded_curve* bcrv = BNDCRV_PTR(c);
    if(bcrv) {
        if(bcrv–>is_line()) {
            Printf(port, ”#[line %x]”,BNDCRV_PTR(c));
        }else if(bcrv–>is_arc()) {
            Printf(port, ”#[circle %x]”,BNDCRV_PTR(c));
        }else {
            Printf(port, ”#[curve %x]”,BNDCRV_PTR(c));
        }
    }else {
        Printf(port, ”#[curve %x]”,BNDCRV_PTR(c));
    }
}
```

Example 3-9.   Printing a Scheme Object

Is Method

Topic: *Scheme Interface

The *is* method returns a logical TRUE if an object is of this type, and it determines the type when some arguments may contain any of several different types. These methods are named is followed by an underscore and the name of the type. Example 3-10 determines if an object is a bounded_curve.

```
// Purpose–––Determine if a Scheme object is a bounded_curve.
```

```
logical is_Scm_Bounded_Curve(ScmObject p)
{
    ENTER_FUNCTION(”is_Scm_Bounded_Curve”);
    return TYPE(p) == T_BndCrv;
}
```

Example 3-10. Determining the Type of a Scheme Object

Get Method

Topic: *Scheme Interface

The *get* method creates a C++ object from a Scheme object of this type. Get methods are named get followed by an underscore and the name of the type. Example 3-11 creates a C++ bounded_curve from a Scheme bounded_curve.

```
// Purpose–––Get a bounded_curve from a Scheme bounded_curve.

bounded_curve* get_Scm_Bounded_Curve(ScmObject crv)
{
    ENTER_FUNCTION(”get_Scm_Bounded_Curve”);
    bounded_curve* bcrv = NULL;
    if(TYPE(crv) == T_BndCrv) {
        bcrv = BNDCRV_PTR(crv);
    }
    if(bcrv == NULL) {
        Wrong_Type_Combination(crv, ”CURVE”);
    }
    return bcrv;
}
```

Example 3-11. Creating a C++ Object from a Scheme Object

Make Method

Topic: *Scheme Interface

The *make* method creates a Scheme object from a C++ object of this type. Make methods are named make followed by an underscore and the name of the type. Example 3-12 creates a Scheme bounded_curve from a C++ bounded_curve.

```
// Purpose–––Create a Scheme bounded_curve from a bounded_curve.
ScmObject make_Scm_Bounded_Curve(bounded_curve* bcrv)
{
    ENTER_FUNCTION(”make_Scm_Bounded_Curve”);
    ScmObject crv = Alloc_Object(sizeof(S_BndCrv), T_BndCrv, 0);
    ((struct S_BndCrv*)POINTER(crv))–>tag = NULL;
    ((struct S_BndCrv*)POINTER(crv))–>The_GC_BndCrv =
        new GC_BndCrv(crv,bcrv);
    return crv;
}
```

Example 3-12. Creating a Scheme Object

Note *In this Example, the* bounded_curve *is no longer accessible from Scheme; the Scheme GC deletes it. Because the curve is not copied, it must not be deleted in the procedure that calls this procedure.*

Scheme Initialization Functions

Topic: *Scheme Interface

A Scheme initialization function is called at program initialization time and does tasks, such as creating required Scheme objects and initializing their contents.

Application developers may create their own Scheme initialization functions depending on the specific requirements of their program. Zero or more initialization functions can be created.

Defining New Scheme Initialization Functions

Topic: *Scheme Interface

The following steps create a new Scheme initialization function:

1. Define an initialization function.

   The function must return void and have no parameters. Refer to the files in the scmext directory for examples of file and function naming conventions.

2. Call the SCM_INIT macro.

   This macro registers the name of the function with the Scheme Interpreter as a function to be called at initialization time. The following is the format of the macro:

   ```
   SCM_INIT(function_name);
   ```

Initialization Function Example

Topic: *Scheme Interface

Example 3-13 shows the initialization for the bounded_curve class described previously in this section.

```
// Purpose---Define S_BndCrv type.

void init_S_BndCrv()
{
    ENTER_FUNCTION(”init_S_BndCrv”);

    T_BndCrv = Define_Type(0,
        ”bounded_curve”,
        (int (*)(size_t))NOFUNC,
        sizeof(S_BndCrv),
        BndCrv_Equal,
        BndCrv_Equal,
        BndCrv_Print,
        (void (*)(ScmObject*, void
        (*)(ScmObject*)))NOFUNC);
}

// Bind the procedure to the Scheme Interpreter so that it is
// called.
SCM_INIT(init_S_BndCrv);
```

Example 3-13. Initializing the bounded_curve Class

The following items are specified in Example 3-13:

init_S_BndCrv	takes no arguments and defines a variable pertaining to bounded curves.
ENTER_FUNCTION	is a diagnostic hook macro commonly used by *Spatial* for stack tracing and other debugging. During normal operations, it does nothing. If the code is compiled with the DEBUG_TRACE flag defined, ENTER_FUNCTION prints the name of the function each time the function is entered.
SCM_INIT	is a binding macro that registers init_S_BndCrv as a function to be run at program initialization time. This macro takes the name of the function as its only argument.

Evaluating Scheme Expressions

Topic: *Scheme Interface

Several evaluator functions exist for use by application developers using or extending the Scheme Interpreter.

Evaluator functions are called from C++. They evaluate (execute) Scheme functions by calling the Scheme Interpreter and then returning the results. They are defined in the sprocess.cxx and elk_eval.cxx files. These functions include the following (refer to the function reference templates for more information):

- SchemeCommand
- SchemeEvaluate
- SchemeGetValue
- SchemeLoad
- SchemeObjectToString
- SchemeSetVariable
- scheme_process
- scm_Funcall
- StringToSchemeObject

Garbage Collection

Topic: *Scheme Interface

Garbage collection is the process of recovering and reusing memory allocated by the Scheme Interpreter. The interpreter never destroys a Scheme object unless it can prove that the object is not used again. It does, however, recycle memory efficiently to prevent excessive memory allocation. This section describes interacting with the Elk Scheme garbage collector.

Making a Scheme Object Known to the Garbage Collector

Topic: *Scheme Interface

Consider the nondestructive version of the primitive shown in Example 3-14, which returns a new vector instead of altering the contents of the original vector.

```
Object p_vector_reverse(Object vec) {
    Object ret;
    int i, j;
    Check_Type(vec, T_Vector);
    ret = Make_Vector(VECTOR(vec)->size, False);
    for (i = 0, j = VECTOR(vec)->size; --j >= 0; i++)
        VECTOR(ret)->data[i] = VECTOR(vec)->data[j];
    return ret;
}
```

Example 3-14. Non–destructive Scheme Primitive p_vector_reverse

In this code a new vector is allocated, filled with the contents of the original vector in reverse order, and returned as the result of the primitive. Make_Vector is declared by Elk as:

```
Object Make_Vector(int size, Object fill);
```

where size is the length of the vector, and all elements are initialized to the Scheme object fill. In the example, the predefined global variable False is used as the fill object; it holds the Scheme constant #f (any Object could have been used here).

Although the C function may look right, there is a problem when it comes to garbage collection. To understand the problem and its solution, it may be helpful to have a brief look at how the garbage collector works. Elk actually employs two garbage collectors, one based on the traditional stop–and–copy strategy, and a generational, incremental garbage collector which is less disruptive but not supported on all platforms.

Note *The following description presents a simplified view; the actual algorithm is more complex.*

In Elk, garbage collection is triggered automatically whenever a request for heap space cannot be satisfied because the heap is full, or explicitly by calling the primitive collect from within Scheme code. The garbage collector traces all ‘‘live’’ objects, starting with a known *root set* of pointers to reachable objects (basically the interpreter’s global lexical environment and its symbol table). Following these pointers, all accessible Scheme objects are located and copied to a new heap space in memory (‘‘forwarded’’), thereby compacting the heap. Whenever an object is relocated in memory during garbage collection, the contents of the pointer field of the corresponding C Object is updated to point to the new location. After that, any constituent objects (e.g., the elements of a vector) are forwarded in the same way.

As live objects are relocated in memory, *all* pointers to an object need to be updated properly when that object is forwarded during garbage collection. If a pointer to a live object were not in the root set (that is, not reachable by the garbage collector), the object would either become garbage erroneously during the next garbage collection, or, if it had been reached through some other pointer, the original pointer would now point to an invalid location. This is exactly what happens in the example shown in Example 3-14.

The call to Make_Vector in the example triggers a garbage collection if the heap is too full to satisfy the request for heap space. As the Object pointer stored in the argument vec is invisible to the garbage collector, its pointer field cannot be updated when the vector to which it points is forwarded during the garbage collection started inside Make_Vector. As a result, all further references to VECTOR(vec) will return an invalid address and may cause the program to crash (immediately or, worse, at a later point). The solution is simple: the primitive just needs to add vec to the set of initial pointers used by the garbage collector. This is done by inserting the line

```
GC_Link(vec);
```

at the beginning of the function before the call to Make_Vector. GC_Link is a macro. Another macro, GC_Unlink, must be called later (e.g., at the end of the function) without an argument list to remove the object from the root set again. In addition, a call to GC_Node (again without an argument list) must be placed in the declarations at the beginning of the enclosing function or block. Example 3-15 shows the revised, correct code.

```
Object p_vector_reverse(Object vec) {
    Object ret;
    int i, j;
    GC_Node;
    GC_Link(vec);
    Check_Type(vec, T_Vector);
    ret = Make_Vector(VECTOR(vec)->size, False);
    for (i = 0, j = VECTOR(vec)->size; --j >= 0; i++)
    VECTOR(ret)->data[i] = VECTOR(vec)->data[j];
    GC_Unlink;
return ret;
}
```

Example 3-15. Scheme Primitive p_vector_reverse, Corrected Version

Any local variable or argument of type Object must be protected in the manner shown above if a function that triggers a garbage collection is called during its lifetime. This may sound more burdensome than it really is, because most of the ‘‘dangerous’’ functions are rarely or never used from within C/C++ extensions or applications in practice. Most primitives that require calls to GC_Link use some function that creates a new Scheme object, such as Make_Vector in the example above.

To simplify GC protection of more than a single argument or variable, additional macros GC_Link2, GC_Link3, and so on up to GC_Link7 are provided. Each of these can be called with as many arguments of type Object as is indicated by the digit (separate macros are required, because macros with a variable number of arguments cannot be defined in C). A corresponding macro GC_Node2, GC_Node3, and so on, must be placed in the declarations. Different GC_Link* calls cannot be mixed. All local variables passed to one of the macros must have been initialized. GC protection is not required for ‘‘pointerless’’ objects such as Booleans (Scheme data type boolean) and small integers, and for the arguments of primitives with a variable number of arguments.

Here is how the implementation of the primitive cons uses GC_Link2 to protect its arguments (the car and the cdr of the new pair):

```
Object P_Cons(Object car, Object cdr) {
    Object new_pair;
    GC_Node2;
    GC_Link2(car, cdr);
    new_pair = allocate heap space and initialize object;
    GC_Unlink;
    return new_pair;
}
```

There are a few pitfalls to be aware of when using ‘‘dangerous’’ functions from within your C/C++ code. For example, consider the following code fragment, which fills a Scheme vector with the program’s environment strings that are available through the NULL terminated string array environ:

```
Object vec = new vector of the right size;
int i;
GC_Node;
GC_Link(vec);
for (i = 0; environ[i] != 0; i++)
    VECTOR(vec)->data[i] = Make_String(environ[i],
    strlen(environ[i]));
}
```

Make_String creates and initializes a new Scheme string. The body of the for–loop contains a subtle bug: depending on the compiler used, the left hand side of the assignment (the expression involving vec) may be evaluated before Make_String is invoked. As a result, a copy of the contents of vec might be, for instance, stored in a register before a garbage collection is triggered while evaluating the right hand side of the assignment. The garbage collector would then move the vector object in memory, updating the properly GC–protected variable vec, but not the temporary copy in the register, which is now a dangling reference. To avoid this, the loop must be modified as follows:

```
for (i = 0; environ[i]; i++) {
    Object temp = Make_String(environ[i], strlen(environ[i]));
    VECTOR(vec)->data[i] = temp;
}
```

A related pitfall to watch out for is exemplified by this code:

```
fragment: Object obj;
...
GC_Link(obj);
...
some_function(obj, P_Cons(car, cdr));
```

Here, the call to P_Cons, just like Make_String above, can trigger a garbage collection. Depending on the C compiler, the properly GC–protected object pointer obj may be pushed on the argument stack before P_Cons is invoked, as the order in which function arguments just like the operands of the assignment operator are evaluated is undefined in the C language. In this case, if a garbage collection takes place and the heap object to which obj points is moved, obj will be updated properly, but the copy on the stack will not. Again, the problem can be avoided easily by assigning the result of the nested function call to a temporary Object variable and using this variable in the enclosing function call:

```
temp = P_Cons(car, cdr);
some_function(obj, temp);
```

Protecting Global Objects

Topic: *Scheme Interface

The previous section explained when and how to register with the garbage collector function. Similarly, global variables must usually be added to the set of reachable objects as well, if they are to survive garbage collections. In contrast to local variables, global variables are only made known to the garbage collector once after initialization, as their lifetime is that of the entire program. To add a global variable to the garbage collector's root set, the macro

```
Global_GC_Link(obj)
```

must be called with the properly initialized variable of type Object. The macro takes the address of the specified object. An equivalent functional interface can also be used:

```
void Func_Global_GC_Link(Object *obj_ptr);
```

This function must be supplied the address of the global variable to be registered with the garbage collector. When writing extensions that maintain global Object variables, Global_GC_Link (or Func_Global_GC_Link) is usually called from within the extension initialization function, immediately after each variable is assigned a value. For instance, the global Scheme vector *handlers* that associate procedures with UNIX signals is initialized and GC–protected as follows:

```
void elk_init_unix_signal(void) {
    handlers = Make_Vector(NSIG, False);
    Global_GC_Link(handlers);
    ...
}
```

NSIG is the number of UNIX signal types as defined by the system include file. The signal handling Scheme procedures that are inserted into the vector later need not be registered with the garbage collector, because they are now reachable through another object which itself is reachable.

Functions That Can Trigger Garbage Collection

Topic: *Scheme Interface

This section lists the functions exported by Elk that may trigger a garbage collection. Within C/C++ code, local Scheme objects must be protected as shown above when one of these functions is called during the object's lifetime.

The C functions corresponding to the following Scheme primitives can cause a garbage collection to occur:

append	load	read–string
apply	macro–body	require

autoload	macro–expand	reverse
backtrace–list	make–list	string
call–with–input–file	make–string	string–>list
call–with–output–file	make–vector	string–>number
call/cc	map	string–>symbol
command–line–args	oblist	string–append
cons	open–input–file	string–copy
dump	open–input–output–file	substring
dynamic–wind	open–input–string	symbol–plist
eval	open–output–file	tilde–expand
for–each	open–output–string	type
force	port–line–number	vector
get–output–string	procedure–lambda	vector–>list
list	provide	vector–copy
list–>string	put	with–input–from–file
list–>vector	read	with–output–to–file
all special forms	*all mathematical primitives*	*all output primitives if output is sent to a string port*

In practice, most of these functions, in particular the special forms,are rarely or never used in extensions or Elk based applications. In addition to these primitives, the following C functions can trigger a garbage collection:

Alloc_Object	Make_Reduced_Flonum	Make_String
Make_Port	Make_Flonum	Make_Const_String
Load_Source_Port	Define_Primitive	Intern
Load_File	Printf	CI_Intern
Copy_List	Print_Object	Define_Variable
Const_Cons	General_Print_Object	Define_Symbol
Make_Integer	Format	Bits_To_Symbols
Make_Unsigned	Eval	Make_Vector

Make_Long	Funcall	Make_Const_Vector
Make_Unsigned_Long		

Make_Integer, Make_Unsigned, Make_Long, and Make_Unsigned_Long can only trigger a garbage collection if FIXNUM_FITS (or UFIXNUM_FITS, respectively) returns zero for the given argument.

A Scheme procedure is a collection of one or more Scheme commands grouped with parentheses. This chapter describes how to write Scheme procedures for use with ACIS.

Procedures are created in a file using a text editor or on the command line while running the interpreter.

Scheme commands can be a mixture of types:

Primitives	Are native to the Elk interpreter.
Extensions	Originate from ACIS and the application developer's own Scheme extensions.
Compounds	Are made of two or more primitives or extensions.