[Lang and Perlmutter 1986] K. Lang and B. Perlmutter. Oaklisp: an Object-Oriented Scheme with First Class Types. Proceedings OOPSLA '86, 30-37, September 1986.

[Maes and Nardi 1987] P. Maes and D. Nardi. *Meta-Level Architectures and Reflection.* North-Holland, Amsterdam, August 1987.

[Manning 1987] C.R. Manning. *ACORE: The Design of a Core Actor Language and its Compiler.* Master's thesis, A.I. Laboratory, M.I.T., August 1987.

[Rees and Clinger 1987] J. Rees and W. Clinger (eds.) *Revised^3 Report on the Algorithmic Language Scheme.*

[Scheevel and Tomlinson] M. Scheevel and C. Tomlinson. *Ceres Language Manual.* MCC Technical Report ACA-ST-145-88(Q), April 1988.

[Snyder 1987] A. Snyder. "Inheritance and the Development of Encapsulated Software Components." in Research Directions in Object-Oriented Languages, Shriver and Wegner eds., 165-188, 1987.

[Sussman and Steele 1975] G. Sussman and G. Steele. *An Interpreter for Extended Lambda Calculus*. MIT AI Memo 349, December 1975.

[Tomlinson 1989] C. Tomlinson and V. Singh. "Synchronization and Inheritance with Enabled-Sets." in Proceedings OOPSLA '89, October 1989.

[Watanabe and Yonezawa 1988] T. Watanabe and A. Yonezawa. "Reflection in an Object-Oriented Concurrent Language." To appear in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), September 1988.

[Yoshida and Chikayama 1988] K. Yoshida and T. Chikayama. *A'UM — A Stream-Based Concurrent Object-Oriented Language.* ICOT Technical Report, 1988.

References

[Agha 1986] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[Agha et al. 1988] G. Agha, W. Kim, M. Scheevel, V. Singh, C. Tomlinson, and R. Will. *Rosette: An Object-Based Concurrent Systems Architecture.* MCC/ACA Technical Report ACA-ST-254-88(Q), August 1988.

[Ametek 1988] *Programmer's Guide to the Series 2010™ System.* Ametek Computer Research Division, preliminary, June 16, 1988.

[Athas 1987] W.C. Athas. *Fine Grain Concurrent Computers.* Ph.D. Thesis, California Institute of Technology, 1987.

[Augustsson 1985] L. Augustsson. Compiling Pattern Matching. Proceedings Functional Programming Languages and Computer Architecture Conference, Springer Verlag LNCS 201, 368-381, September 1985.

[Bawden and Rees 1988] A. Bawden and J. Rees. Syntactic Closures. Proceedings Lisp and Functional Programming Conference 1988, 86-95, July 1988.

[Delagi et al. 1987] B.A. Delagi, N. Saraiya, S. Nishimura, and G. Byrd. An Instrumented Architectural Simulation System. STAN-CS-87-1148, Computer Science Department, Stanford University, January, 1987.

[Dijkstra 1975] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1975.

[Dybvig and Hieb 1988] K. Dybvig and R. Hieb. A Variable-Arity Procedural Interface. Proceedings Lisp and Functional Programming Conference 1988, 106-115, July 1988.

[Encore 1985] *Multimax technical summary.* Encore Computer Corporation, May 1985.

[Gabriel et al. 1988] R. Gabriel et al. *Common Lisp Object System.* ANSI X3J13 Document 87-002.

[Gelernter 1985] D.H. Gelernter. "Generative Communication in Linda." ACM Transactions on Programming Languages and Systems, 80-112, Jan. 1985.

[Goldberg and Robson 1983] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1983.

[Hoare 1984] C.A.R. Hoare, ed. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Kohlbecker 1986] E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* U. Indiana Dept. Comp. Sci. Technical Report 199, August 1986.

[LaLonde et al. 1986] W.R. LaLonde, D.A. Thomas, and J.R. Pugh. "An Exemplar Based Smalltalk." Proceedings 1986 OOPSLA, 322-330.

with a suffix of **.rbl** concatenated. If the **'silent** option is given then no results are displayed during loading.

(repl) ⇒ **#niv** procedure
(reset) ⇒ **#niv** procedure
(rosette) ⇒ **#niv** procedure
(kernel) ⇒ **#niv** procedure

Interaction with the Rosette system is mediated by a listener-loop. The usual listener-loop is invoked by the **rosette** procedure. This loop uses the syntax expander on all expressions read in. The prompt used in this loop is: **rosette>**. The **rosette** procedure sets **repl** to the Rosette repl procedure so that **reset** will restart the Rosette listener-loop. The **reset** procedure is invoked by the virtual machine when it is interrupted by the user and on handling errors. The procedure **kernel** is similar to **rosette** but enters the primitive listener supported directly by the virtual machine. Its prompt is **kernel>**. The most significant difference between the two listeners is that the **kernel** does not call **expand** on expressions that are read in, thus none of the derived syntax forms are available. If an error occurs that terminates the **rosette** listener and puts the machine in the **kernel** loop then a **reset** will usually solve the problem.

(read *Istream***)** ⇒ *expression or* **#eof** operation

This operation reads a single expression from the stream or returns **#eof** if at the end. If the **read** operation detects a syntax error then it returns the constant **#eof** and sets the read-state of the *Istream* to failed.

(state *Istream***)** ⇒ **0 ... 7** operation
(failed? *Istream***)** ⇒ *Boolean* operation
(clear *Istream***)** ⇒ *Istream* operation

The **state** operation returns the read-state of the stream. **0** signifies a good read. If the result includes the **1** bit then an end-of-file condition has occurred. If the result includes the **2** bit then the Rosette reader has signalled a failed read. This is typically due to a syntax error on input. If the result includes a **4** bit then some other form of corruption has occurred on the stream. The **failed?** operation is a convenient way of determining whether the reader detected a syntax error or not. If a read fails then the stream should be cleared with the **clear** operation. If the read-state is not good or failed then the stream should be closed. Clearing a stream that is at end-of-file or bad and then attempting to **read** will cause a non-recoverable error.

(print *items***)** ⇒ **#niv** operation
(print *Ostream items***)** ⇒ **#niv** operation
(print\n *items***)** ⇒ **#niv** operation
(print\n *Ostream items***)** ⇒ **#niv** operation
(display *items***)** ⇒ **#niv** operation
(display *Ostream items***)** ⇒ **#niv** operation

These operations are the basic means of performing output. Each are variadic and may be applied directly to an output stream or not. If no output stream is used then **stdout** is assumed. **print** and **print\n** include quotes on strings, expressions and symbols where **display** does not. **print\n** finishes printing with a new-line.

(flush *Ostream***)** ⇒ **#niv** operation

The **flush** operation ensures that all output to the stream has reached the destination file.

(load *file_id* **& 'silent)** ⇒ **#niv** procedure

The **load** procedure provides for reading in and expanding, compiling, and running a file containing Rosette expressions. The *file_id* may be either a string or a symbol. If the *file_id* is a fully qualified UNIX pathname then **load** uses it directly signalling an error if the named file cannot be opened. If the *file_id* is not fully qualified then a search process is initiated. If the **load** occurs within a file being loaded then the directory from which the load is occurring is searched first, next the current working directory is searched, and finally the tuple of paths in the variable **load-paths** is searched for *file_id* that defines how to expand the request expression. Most syntax extensions in Rosette are implemented in just this way. In order to facilitate these definitions the following form is provided:

(defExpander (*id id'*) *body*) ⇒ '*id* derived syntax

The **defExpander** syntax defines the method, the anonymous operation, and other bookkeeping associated with introducing *id* as a new syntactic keyword. The *id'* is whatever the macro writer wants to call the expander argument that will be passed when the new expander method is invoked. The expander operation bound to *id'* is typically used to expand sub-structure and is itself passed on in the recursive descent. The method that is built by **defExpander** is defined in the context of the prototypical **RequestExpr** so that the **trgt** and **msg** slots may be referred to directly in the *body*. Essentially, **defExpander** defines an extension to the behavior of **RequestExpr**s. The expander for **and** (see section **7.2**) may be written:

```
(defExpander (and e)
  (cond ((same? msg '[])  '#t)
        ((= (size msg) 1) (e (head msg) e)
        (else             (new IfExpr
                               (e (head msg) e)
                               (e (new RequestExpr 'and (tail msg)) e)
                               #f))))
```

The first clause of the **cond** defines the case of **(and)** as simply **#t**. The second clause says that the result of **(and** *expr***)** is simply the result of *expr*. In this case, the **(head msg)** is the single *expr* and it should be expanded with whatever expander is bound to **e**. The **else** case applies when there are two or more conditions being anded. In these cases, a new **IfExpr** is built with the expansion of the **(head msg)** as the condition of the **IfExpr**, the expansion of a new **and** form on the **(tail msg)** for the then-branch, and **#f** for the else-branch. Thus if the expression resulting from **(head msg)** evaluates to **#f** then the **and** terminates immediately with **#f**; otherwise, process continues with the next argument to the **and**.

11. Input and output

This section defines some basic facilities for performing input and output on streams. The streams initially defines are named **stdin**, **stdout**, and **stderr**.

(new *Istream_or_Ostream file_id* **&** *how***)** ⇒ *Istream_or_Ostream* operation

The **new** operation creates a new stream of the appropriate kind.

(close *stream***)** ⇒ **#niv** operation

The **close** operation closes the *stream*. It should be noted that streams are also closed by the virtual machine when they are garbage collected so it is not strictly necessary to explicitly close a stream.

9.4. Error operations

The error operations are synchronous operations that are applied to various entities by the virtual machine or other Rosette code. They are used to signal that certain exceptional conditions have occurred. In Rosette 1.0 the context that was in error can not be continued but useful information can be displayed. The error operations are: **`runtime-error`**, **`missing-method`**, **`formals-mismatch`**, and **`vm-error`**. The **`runtime-error`** is issued on most errors and with future releases will be replaced by finer grained error operations. The **`missing-method`** error occurs exactly when a message is received and no method can be located. The **`formals-mismatch`** occurs when a pattern has been violated in the template for a procedure, method, let, or letrec. The **`vm-error`** occurs if an out-of-bounds reference is made by a lexical lookup byte-code. This is indicative of either a compiler error, or a mismatch between the compiled code for a method and the layout of the map for an entity.

10. Syntactic extensions and expression actors

A syntactic extension mechanism is provided in Rosette so that more convenient forms of expression can be added to the basic syntax of the language. All derived syntax forms are implemented via syntactic extensions defined in Rosette. Syntactic extensions are accomplished by writing expanders that transform expressions into other expressions. In Rosette, expressions are objects in their own right and are written as literals via the quote `'` (see sections **4.1** and **8.1**). This section provides a brief discussion of the syntax extension (macro definition) facilities of Rosette.

`(expand` *expr***`)`** ⇒ *expr′* operation

The **`expand`** operation recursively descends through the *expr* transforming sub-structure as it goes. The syntax expansion model is called expansion-passing style after the concept of continuation-passing in Scheme. The **`expand`** operation actually performs the following request on *expr*:

`(expander` *expr* **`expander)`**

There is a method associated with the **expander** operation on every kind of expression and a default on all other entities that is the identity. Every expansion method is passed the “expander” that it should use to expand sub-structure. This is why **`expander`** is passed to **`expander`** along with the *expr*. During expansion other expander operations may be passed in order to achieve context-sensitive expansion of expressions. The interesting part of syntax expansion occurs when **`expander`** is applied to a request expression. The method associated with **`expander`** on **`RequestExpr`** asks the object in its **`trgt`** slot to return a “target expander” operation that is to be used to expand the request expression. The case of interest is when the **`trgt`** slot contains a symbol that is a syntactic keyword such as **`and`**. When this occurs an (anonymous) operation is returned that is bound to a method on **`RequestExpr`**s tity's **`mbox`** is the **`EmptyMbox`**, then the entity has no messages waiting, is not locked, and will accept the next message received (i.e. the enabled-set is null). The **`mbox`** for all of the non-extensible built entities is always the **`EmptyMbox`**. The **`LockedMbox`** signifies that the entity is busy processing a message and has no messages waiting. If an entity has a non-null enabled-set or has messages waiting to be accepted then it will have **`QueueMbox`**. This kind of mailbox has three slots: **`'locked`**, **`'enabled-set`**, and **`'queue`**. The **`locked`** slot is **`#t`** if the actor is busy and **`#f`** otherwise. The **`enabled-set`** slot holds the current enabled-set for the entity, and the **`queue`** slot refers to a **`Queue`** object that holds the messages waiting to be accepted.

`(messages` *any***`)`** ⇒ *tuple* sync-operation

This operation accesses the **`mbox`** of *any* and returns a tuple of the messages waiting to be accepted by *any* in the order in which they arrived. It is written in Rosette and can be helpful in debugging concurrent programs.

9.3. Monitors

A **`Monitor`** is an object that can be passed to the **`run`** primitive to monitor the time and resource usage of a collection of entities. The monitor may be created with a procedure that will be called at each message step. Such a procedure is given the message step number and the number of actors waiting to execute in the message step. The procedure will typically display this information or write it to a file where it may be analyzed off-line. A monitor may be "converted" through the primitive **`monitor-convert`**. The result is a tuple consisting the result of the **`run`** under which the monitor was used; a triple of the user, gc, and system times; a pair consisting of the number of bytecodes executed and the raw histogram of bytecode usage; a pair consisting of the number of entities created and the raw histogram of entity creations; a pair giving the starting and ending message step numbers for the computation, and a pair giving the maximum number of actors ready at any step and the total number of messages received. Two procedures are provided to make it more or less convenient to use these facilities.

`(time` *expr* **`&`** *sentinel***`)`** ⇒ *tuple* procedure
`(std-sentinel` *Ostream***`)`** ⇒ **`#niv`** procedure

The **`time`** procedure is given an expression to monitor and an optional sentinel procedure. The *tuple* result of **`time`** is of the form above for **`monitor-convert`**. Future releases of Rosette will include a more friendly interface than a raw tuple of information. The **`std-sentinel`** procedure will create a sentinel that will display the message step and activity at each step on the *Ostream*.

for the entity so that the meta information held in the original meta object is inherited by the new meta object. The new meta object only has the basic three slots that are required of any object.

The **extensible** slot is either **#t** or **#f** corresponding to whether entities described by the meta object may have slots added or not. For space and performance considerations, many kinds of objects do not permit new slots to be added. For example, **Fixnums** are represented in the same space as a reference to an actor and there is really no place to put additional slots. The entities that permit new slots to be added are **Actors**, **Operations**, and **Metas**. All others are not **extensible**.

(lookup-obo (meta *any***)** *any key***)** ⇒ *any* — primitive
(get-obo (meta *any***)** *any key***)** ⇒ *any* — primitive

The **lookup-obo** operation asks the **(meta** *any***)** to lookup the *key* on behalf of *any*. This operation performs **get-obo** on *any* and if the slot is defined among the object slots of *any* then its binding is returned; otherwise, the process continues with the **parent** of *any*. If the entity passed as **(meta** *any***)** is not in fact *any*'s meta then a **runtime-error** is signalled.

(add-obo (meta *any***)** *any key any***')** ⇒ *any* — primitive

This operation asks the meta of an entity to add a new object slot with key *key* and value *any***'**. If the meta is marked as **extensible** a reference to *any* is returned from the operation; otherwise, a **runt-ime-error** is signalled.

(set-obo (meta *any***)** *any key any***')** ⇒ *any* — primitive

This operation permits any object slot of any entity to be altered by appealing to its meta object. The object being modified is returned as the result of **set-obo**.

(contour (meta *any***)** *any***)** ⇒ *tuple* — primitive

The **contour** primitive returns a *tuple* consisting of the pairs of keys and values for the object slots of *any*.

```
(contour (meta '(f x)) '(f x))          ⇒ ['trgt 'f 'msg '[x]]
```

(keys (meta *any***)** *any***)** ⇒ *tuple* — primitive

The **keys** primitive returns a tuple of the keys defined on *any*.

```
(keys (meta '(f x)) '(f x))             ⇒ ['trgt 'msg]
```

9.2. Mailboxes

The mailbox objects implement the communications interface of an entity. The main function of a mailbox is the buffering of incoming messages until they are accepted by the entity for processing. There are three kinds of mailboxes provided in Rosette: **EmptyMbox**, **LockedMbox**, and **QueueMbox**. If an en-

There are primitive operations for accessing the actors bound to the meta slots of an entity:

(parent *any***)** ⇒ *any* — primitive
(meta *any***)** ⇒ *meta* — primitive
(mbox *any***)** ⇒ *mbox* — primitive

Each of these primitive operations is defined on every entity in the Rosette system. There are corresponding primitive operations that permit the bindings of the meta slots of an entity to be altered. They are named by suffixing a colon (`:`) to each of the above primitive names. These operations are used in bootstrapping the Rosette system.

This remainder of this section will describe the **`meta`** and **`mbox`** objects. The **`parent`** objects where discussed in section **8.5**. This section will also describe the monitoring facilities and error operations provided in Rosette.

9.1. Meta objects

The meta object describes the structure of an entity and implements the operations that access the object slots of an entity. Each meta object contains at least three slots: **`'map`**, **`'ref-count`**, and **`'extensible`**. The **`map`** slot contains a table that gives the correspondence between the keys defining the object slots of an entity and the offset within the representation of the entity at which the slot occurs. Thus every entity is conceptually a contiguous chunk of storage which is big enough to hold a reference to the binding for each of its object slots as well as its three meta slots. The keys are held in the meta object which is in general shared among all of the entities with the same description. This means that the space required to maintain the mapping of keys to values need only occur once for all similar entities and that the compiler can make use of the offset information to compile fast access to slots without requiring associative key lookup at runtime. Thus, there are typically far fewer meta objects than entities in a system. Further, the system starts off with a meta object that describes itself so there is no infinite regress of meta objects. Meta slots may not be added to any entity; however, object slots may be added to the meta object itself of any entity. This permits information about an entity to be captured in its corresponding meta object. Most of the printing system is implemented in terms of methods on the meta objects of the various entities in Rosette.

The **`ref-count`** slot counts the (maximum) number of objects described by the meta object. If it is **`0`** then there are no objects described, if it is **`1`** then the association between entity and meta object is 1-1, and if it is greater than **`2`** then we say that the meta object is shared. Adding slots to an entity whose meta object is shared among other entities requires that a new meta object be created for the entity to which a slot is being added. The **`parent`** of the new meta object is made to be the original meta object

9. System Actors

The Rosette execution model [Agha et al. 1988] defines the resources that support an actor in terms of parents, meta objects, and mailbox abstractions so that resource management decisions may be expressed at the level of individual application actors. There are two major roles for resource abstractions. The first is to provide the implementation of application actors, and the second is to provide monitoring and control interfaces so that resource management policies may be programmed to monitor and control performance. Many control and management abstractions may be expressed in terms of the execution model — for example, transactions and actor migration. Further, monitoring and debugging facilities in a system are programmed in terms of the execution model.

The structural model of an actor represents an actor in terms of other actors (see Figure 2.). The structure of an actor may be thought of as a set of slots, each of which is a pair consisting of a key and a value. There are conceptually two classes of slots associated with every actor: the "meta" slots and the "object" slots. The object slots contain bindings for values (including methods) that are components of the state and behavior of the kind of thing being modeled by an actor. The meta slots contain references to other actors that implement an actor. There are three meta slots defined for every actor: **`parent`**, **`meta`**, and **`mbox`**. The actors bound to these slots are collectively termed *resource actors*. Any combination of these resource actors may be built-in or not; however, there must be at least one set of primitive resource actors that define a concrete implementation of actors in a system. It is the possibility of defining new resource actors within the context of the execution model that makes the Rosette base language extensible so that new forms of monitoring and control may be defined.

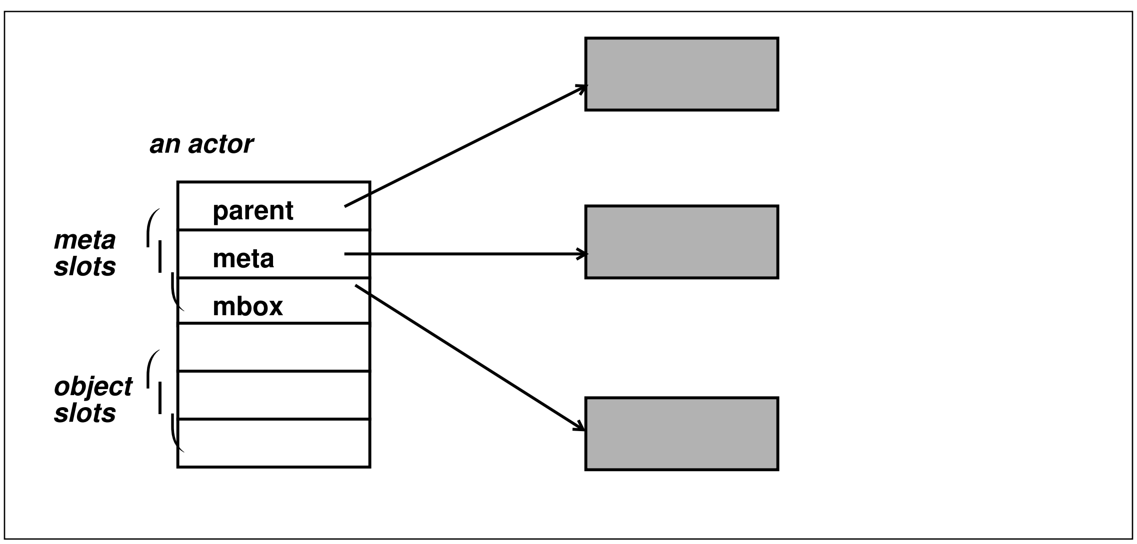

Figure 2: Structural model of an actor

8.6. Operation objects

Operations are typically defined in the **`Global`** environment by the **`defOprn`** and **`defSync`** forms described in section **6.2**. Anonymous operations may be created by the **`new`** operation on **`Oprn`** and **`SyncOprn`**. Operation objects are specializations of **`Actor`** and as such new slots and methods may be added; however, all operations come equipped initially with two slots : **`'id`** and **`'sync`**. The **`id`** slot typically holds a symbol that identifies the operation and the **`sync`** slot holds a boolean that if **`#t`** indicates that the operation is a synchronous operation and otherwise indicates that it is an ordinary operation. When a synchronous operation is used with a target object or actor, then the process of looking up the operation and invoking the binding is performed without regard for whether the entity is locked or not. Non-synchronous operations obey the basic actor semantics in that when sent to a locked entity the message is enqueued in the mailbox of the entity. Synchronous operations are used to implement protocols that aid in debugging, browsing, and initialization.

8.3. Methods

A method object holds the message pattern and body that defines the actions taken in response to a message received by an actor or object. It is an unclosed procedure in that it does not carry an environment with it. The environment in which a method runs is determined by the object or actor receiving a message that invokes the method. Method objects contain three slots: **'code**, **'id**, and **'source**. The **code** slot contains a code object as described above. The **id** slot typically contains a symbol that corresponds to the operation or local identifier that is used to invoke the method, and the **source** slot holds the expression that was compiled to produce the method object. The **source@**, **formals@**, **code@**, and **show-code** operations are defined on method objects (see section **4.3.2**).

8.4. Procedure objects

Procedure objects are similar to method objects but in addition they carry their own environment in which they run when they are sent a message. The slots for a procedure object are: **'env**, **'code**, **'id**, and **'source**. The **env** slot holds the object that represents the environment over which the procedure was closed. It is this environment that will be extended with the bindings defined by the template and pattern in the procedure object's code. As with the method object, the operations **source@**, **formals@**, **code@**, and **show-code** are defined on procedure objects.

8.5. Shared Behavior and Multiple Inheritance objects

A shared behavior object is just an ordinary object (or actor) that acts as a repository for methods and values shared across a family of instances. By convention in Rosette, actors used to hold shared behavior are identified as such and the defining forms such as **defActor** and **defMethod** depend on these conventions. Actors have a single meta slot, accessed via the **parent** primitive operation, that is used to support inheritance during the process of looking up a binding to an identifier or operation. Thus, single inheritance is accomplished simply by linking objects together via the **parent** slot in the order that is desired. The environment extensions that occur during procedure and method invocation just link more objects along the inheritance chain so that lexical environments and single inheritance are integrated in a single mechanism. In order to support multiple inheritance, a specialization of **Actor** is included in Rosette. The prototype in the **Global** environment is named **MIObject** for Multiple Inheritance Object. While **MIObject**s may be extended with additional slots, they each come equipped with a slot named **'cpl** for class precedence list. This is a tuple of (shared behavior) objects that are searched in linear order for the desired key. If the **cpl** is empty, the **MIObject** returns **#absent** as the result of the lookup operation. If the **cpl** is not empty but the key is not present in any of the objects of the **cpl** then the search continues with the **parent** of the last object in **cpl**.

The let and letrec expression prototypes are named respectively: **`LetExpr`** and **`LetrecExpr`**. Each kind of expression objects has two slots: **`'bindings`** and **`'body`**. The **`bindings`** is a tuple expression consisting of tuple expressions representing each of the bindings.

Prototypes for procedure and method expressions are named **`ProcExpr`** and **`MethodExpr`** respectively. Each kind of expression object has the following slots: **`'id`**, **`'formals`**, and **`'body`**. The **`formals`** is a tuple expression representing the pattern of messages that are accepted.

For each of the label, goto, and set! expressions, there are prototypes named: **`LabelExpr`**, **`GotoExpr`**, and **`SetExpr`**. The slots for a label expression are: **`'label`** and **`'body`**. The goto expression has a single **`'label`** slot and set! expression has a **`'trgt`** slot for the lexical binding to be set and a **`'val`** slot to hold the expression that will be evaluated to produce the value for the set! expression.

8.2. Code, Templates and Patterns

Code objects result from the **`compile`** operation on expressions. Typically, a code object corresponds to an expression entered to the Rosette listener or a procedure or method expression. Code objects contain a **`'codevec`** slot and a **`'litvec`** slot. The **`codevec`** is a sequence of byte-codes and the **`litvec`** is a tuple of literals referred to from the **`codevec`**. In the case of procedure and method objects, the **`litvec`** always contains a template object. Templates control the destructuring and binding of formals according to the pattern object that is contained in a template object. Pattern objects represent the patterns of messages that are accepted when procedures and methods are invoked. Pattern objects have an **`'expr`** slot that holds a tuple expression representing the pattern. The prototypes for the various kinds of patterns in the **`Global`** environment and the bindings for their **`expr`** slot are:

```
IdVecPattern                      ⇒ '[x]
IdAmperRestPattern                ⇒ '[& r]
ComplexPattern                    ⇒ '[[a b] c]
```

Code, template and pattern objects are created by the **`compile`** operation on procedure and method expression objects. The operations **`source@`**, **`formals@`**, and **`show-code`** are defined on code objects.

`(run` *code* **`&`** *monitor***`)`** ⇒ *any* primitive

The **`run`** primitive causes the virtual machine to execute the *code*. An optional *monitor* may be provided that will accumulate statistics on the execution of the *code*. Monitors are discussed in section **9**. The result of the **`run`** request is the result of executing the *code*.

(new* TupleExpr *sequence_of_expressions*) ⇒ *tuple_expression* — operation

The **new*** operation creates tuple expressions from their constituent expressions.

(->tuple *tuple_expression*) ⇒ *tuple_of_expressions* — operation

The **->tuple** of a tuple expression is the tuple of expressions preceding the **&** (if any) in the tuple expression.

```
(->tuple '[a (+ 1 b)])          ⇒ ['a '(+ 1 b)]
(->tuple '[& (foo 3)])          ⇒ []
(->tuple '[])                   ⇒ []
```

(split *tupleExpr number*) ⇒ [*expr* … *tupleExpr*] — operation

The **split** operation returns a tuple consisting of the first *number* sub-expressions of *tupleExpr* and a final element consisting of the remainder of the *tupleExpr*.

```
(split '[a b c & r] 1)          ⇒ '['a '[b c & r]]
```

(size *tupleExpr*) ⇒ *number* — operation
(nth *tupleExpr number*) ⇒ *expression* — operation
(sub-obj *tupleExpr number number*) ⇒ *tuple_expression* — operation

The **size**, **nth**, and **sub-obj** operations are defined on the prefixes of tuple expressions. The **size** of a tuple expression is the size of the prefix of the expression. The use of the **nth** and **sub-obj** operations coincides with that for tuples (section **7.6**).

```
(define m1 '[(+ 1 2) (* 3 4) (- 5 6)])
(define m2 '[(+ 7 8) & m1]
(size m1) ≡ (size (->tuple m1))     ⇒ 3
(size m2) ≡ (size (->tuple m2))     ⇒ 1
(nth m1 1)                          ⇒ '(* 3 4)
(nth m2 1)                          ⇒ runtime-error
(sub-obj m1 1 2)                    ⇒ '[(* 3 4) (- 5 6)]
(sub-obj m2 0 1)                    ⇒ '[(+ 7 8)]
```

8.1.4. Other expression objects

The prototype representing quote expressions is named **QuoteExpr**. A quote expression has a single slot named **'expr** that holds the expression being quoted.

The prototype representing free expressions is named **FreeExpr**. Free expressions have two slots: **'id-list** and **'body**. The **id-list** is a tuple expression consisting solely of symbols.

The prototype if expression is named **IfExpr**. If expressions have three slots: **'condition**, **'then-branch**, and **'else-branch**. The **else-branch** may be filled with the **EmptyExpr** for one-armed ifs.

(new* *block_or_seqExpr sequence_of_expressions*) ⇒ *block* — operation

The **new*** operation creates a block or sequential expression from its constituents, and is convenient when the constituent expressions are not already in a tuple.

(size *block_expression*) ⇒ *number* — operation
(nth *block_expression number*) ⇒ *expression* — operation
(sub-obj *block_expression number number*) ⇒ *block_expression* — operation

The **size**, **nth**, and **sub-obj** operations are defined on these two kinds of expressions. The use of the **nth** and **sub-obj** operations coincides with that for tuples (section **7.6**).

```
(define b '(block (+ 1 2) (* 3 4) (- 5 6)))
(size b)                                  ⇒ 3
(nth b 1)                                 ⇒ '(* 3 4)
(sub-obj b 1 2)                           ⇒ '(block (* 3 4) (- 5 6))
```

8.1.2. Communication expression objects

There are two kinds of communication expressions: **RequestExpr**, and **SendExpr**. Each of these expression objects has two fields: **'trgt** and **'msg**. The **msg** is typically a tuple expression object, a symbol, or a request expression object.

(new* *request_or_sendExpr expression sequence_of_expressions*) ⇒ *request_expression* — operation

The **new*** operation creates the corresponding kind of communication expression from its components.

```
(new* RequestExpr '+ '1 '2)               ⇒ '(+ 1 2)
(new* SendExpr 'foo 'bar 'baz)            ⇒ '(send foo bar baz)
```

(split *requestExpr number*) ⇒ **[***expr … tupleExpr***]** — operation

The **split** operation is useful in write syntax expanders. The operation returns a tuple consisting the first *number*+1 elements of which are the **trgt** of the *requestExpr* and the first *number* elements of the **msg**. The last element of the result tuple is the *tupleExpr* remaining in the **msg** after removing its first *number* elements.

```
(split '(f a b c & r) 1)                  ⇒ '['f 'a '[b c & r]]
```

8.1.3. Tuple expression objects

The prototypical **TupleExpr** represents expressions that when evaluated produce a tuple. A tuple expression can be considered to consist of a prefix and an expression for the rest of the tuple that will result when the expression is evaluated. The accessor operations on tuples are defined directly on tuple expressions to facilitate program transformations. These operations are defined on the prefix of the expression.

8. Program Representation Objects

This section discusses the operations available on the objects that are used to represent Rosette programs. These are expressions, patterns, methods, procedures, shared behavior objects, and operations.

8.1. Expression objects

As discussed in previous sections, there are distinct kinds of objects for each of the different basic expression forms. This is how programs are represented as data to other Rosette programs. Strictly speaking, **`Symbol`**s should be considered as identifier expression objects, but they are treated separately since they have other uses.

(expand *expr***)** ⇒ *expr′* — operation
(compile *expr* **&** *any***)** ⇒ *code* — operation

All expressions (including **`Symbol`**s) may be **`expand`**ed and **`compile`**d. The **`expand`** operation returns the result of syntax expansion on *expr* as *expr′* (see section **10**).

```
(expand '(and (f x) (g y)))            ⇒ '(if (f x) (g y) #f)
```

The **`compile`** operation returns a *code* object that may be **`run`** to obtain the effect and value of the *expr*. The *expr* may be directed to **`compile`** itself with respect to an arbitrary entity. This allows any free variables such as references to slots of an object to be resolved at compile time.

(new *expr-object args***)** — operation

Each of the different kinds of expressions can be created via the **`new`** operation by giving an instance of the kind of expression that is to be created and the *args* necessary to fill the slots of the particular kind of expression:

```
(new RequestExpr 'a-trgt 'msg)         ⇒ '(a-trgt & msg)
(new BlockExpr ['(+ 1 2) '(update)]    ⇒ '(block (+ 1 2) (update))
```

The remainder of this section presents information specific to each of the different kinds of expression objects. In particular, the values that need to be provided in the **`new`** operation are detailed. Also the names of their slots are given so that they may be referred to in methods that extend the behavior of expression objects.

8.1.1. Block and Seq expression objects

The prototypical block expression is named **`BlockExpr`** and the prototypical sequential expression is named **`SeqExpr`**. Each block or sequential expression has a slot which is a tuple of expressions. The name of this slot is **`'sub-exprs`**. Thus, a single tuple of expressions is needed when using **`new`**.

ment has been visited. Usually, the *proc* performs some actions for effect, such as displaying the element on a stream:

```
(walk [1 2 3 4] (proc [i x] (print x #\ ))) prints
1 2 3 4
```

7.7. Symbols

Symbols are atomic objects that have an attribute called the name of the symbol. They are objects that represent expressions consisting solely of an identifier. Two symbols are identical (**`same?`**) if and only if their names are spelled the same way (including case). This is exactly the property needed to represent identifiers in programs represented as data. Symbols may also be used as enumerated values as in Pascal. A symbol may be written as an Rosette identifier preceded by a **`'`** (e.g., **`'an-identifier`**).

`(-> Symbol)` ⇒ *operation* — operation

The prototype **`Symbol`** responds to the **`->`** operation with an operation **`->symbol`** that may be used on other kinds of actors to produce a symbol corresponding to the actor. In particular this is used to generate symbols from strings — for example, concatenation of a prefix with a unique number.

```
((-> Symbol) (concat "Gen" (->string 123)))
                                          ⇒ 'Gen123
```

`(->string` *symbol***`)`** ⇒ *string* — operation

The operation **`->string`** when sent a symbol will respond with a string that is unique to the symbol (sometimes called its print-name).

```
(->string 'aBcD)                          ⇒ "aBcD"
```

(concat *tuple tuple …*) ⇒ *tuple* operation

The operation **concat**, when sent two or more tuples, returns a tuple that is the concatenation of its arguments.

```
(concat [1 2 3] [4 5 6])              ⇒ [1 2 3 4 5 6]
```

(nth *tuple index*) ⇒ *any* operation

The **nth** operation expects a tuple and an index and returns the n^{th} element of the tuple.

```
(nth [2 3 4 5 6] 3)                   ⇒ 5
```

(set-nth *tuple index any*) ⇒ *tuple* operation

This operation sets the *index* element of *tuple* to *any* and returns the original *tuple*, modified at the *index* position.

(head *tuple*) ⇒ *any* operation

This operation is equivalent to **(nth** *tuple* **0)**.

(sub-obj *tuple index size*) ⇒ *tuple* operation

The operation **sub-obj** when given a tuple and two numbers returns a (sub) tuple *index*ed from the first number, and has a *size* equal to the second number.

```
(sub-obj [1 2 3 4 5] 2 3)             ⇒ [3 4 5]
```

(tail *tuple*) ⇒ *tuple* operation

This operation is equivalent to **(sub-obj** *tuple* **1 (- (size** *tuple*) **1))**.

(map $tuple_1$ *proc* **&** *args*) ⇒ $tuple_2$ operation

This operation creates a new $tuple_2$ by sending each element of $tuple_1$ and its corresponding index to the *procedure* in a request and placing the reply in the corresponding position of $tuple_2$.

```
(map [1 2 3 4] (proc [i v] (+ v 2)))       ⇒ [3 4 5 6]
```

If any *args* are included they are passed in each call to the *proc*:

```
(map [1 2 3 4] (proc [i v x] (+ v x)) 2)  ⇒ [3 4 5 6]
```

(walk *tuple proc* **&** *args*) ⇒ *tuple* operation

The **walk** operation visits each element of *tuple* calling the *proc* with the index of the element, the value of the element and any *args* that may be supplied. The **walk** returns the original *tuple* after each ele-

The operation **concat** can be used to create a new string from two or more strings:

```
(concat "A " "word")                    ⇒ "A word"
```

(nth *string number***)** ⇒ *character* operation

The **nth** operation applied to a string returns the n^{th} character of the string 0-relative:

```
(nth "A word" 2)                        ⇒ #\w
(nth "A word" 1)                        ⇒ #\
(nth "A \"" 2)                          ⇒ #\"
```

(sub-obj *string number number***)** ⇒ *string* operation

The operation **sub-obj** can be used to obtain a substring of a given string as specified by a 0-relative index and a size:

```
(sub-obj "A word" 2 4)                  ⇒ "word"
```

(->symbol *string***)** ⇒ *symbol* operation

These operations convert strings respectively to numbers and symbols, and are principally used in input operations and construction of new symbols in programs.

7.6. Tuples

Tuples result from evaluating expressions of the form `'[`⟨*expr*⟩ `...]` and performing operations on tuples. Tuples are sequences of elements that occur in a particular order. Tuples are indexed 0-relative and thus the maximum index is **(- (size** *tuple***) 1)**. A **runtime-error** is signalled on out-of-bound indices.

(new Tuple *sequence_of_values***)** ⇒ *tuple* operation
(newN Tuple *fixnum expr***)** ⇒ *tuple* operation

The **new** operation will return a newly created tuple actor. **newN** returns a tuple of size *fixnum* which is initialized so that each element is a copy of the result of *expr*.

```
(new Tuple)                             ⇒ []
(new Tuple 1 'a "a")                    ⇒ [1 'a "a"]
(newN Tuple 4 'a)                       ⇒ ['a 'a 'a 'a]
```

(size *tuple***)** ⇒ *number* operation

The **size** operation returns the number of elements in the tuple.

```
(size [])                               ⇒ 0
(size [2 4 6 8])                        ⇒ 4
```

(null? *tuple***)** ⇒ *boolean* operation

The predicate **null?** returns **#t** if the *tuple* has **size 0**, and returns **#f** otherwise. This operation is equivalent to an explicit test for **(= (size** *tuple***) 0)** or **(same?** *tuple* **[])**.

The following familiar numeric operations are defined on both kinds of numbers:

(+ n1 n2 …)	**(- n1 n2)**	**(* n1 n2 …)**
(/ n1 n2)	**(min n1 n2 …)**	**(max n1 n2 …)**
(abs n1)	**(expt n1 n2)**	

In addition, the following are defined on **Fixnums**:

(% n1 n2)	**(cdiv n1 n2)**	
(lg n1)	**(lgf n1)**	
(lognot n1)	**(logand n1 n2 …)**	**(logxor n1 n2 …)**
(logor n1 n2 …)	**(->float n1)**	

Also, the following are defined on **Floats**

(floor n1)	**(ceil n1)**	**(exp n1)**
(log n1)	**(log10 n1)**	**(->fixnum n1)**

(->string *number***)** ⇒ *string* operation

Numbers respond to the coercion operation **->string**:

```
(->string 3291)        ⇒ "3291"
(->string -624)        ⇒ "-624"
(->string 12.3e-2)     ⇒ "0.123"
```

(->char *number***)** ⇒ *character* operation

Fixnums also respond to **->char**:

```
(->character 65)       ⇒ #\A
(->character 933)      ⇒ #\%
```

7.5. Strings

Strings are sequences of characters, indexed from **1** to **(size** *string***)**. They are written as sequences of characters enclosed within matching " characters. The character " can be included in a string by preceding it with a \. The \ may be also be included by preceding it with a \. A string may be empty "", and has **size 0**.

"A string with a \" and a \\ in it" *displays*

A string with a " and a \ in it

(-> String) ⇒ *operation* operation

The operation for converting objects to strings, named **->string** in the initial environment, provides an interface so that a displayable representation of actors may be uniformly generated.

(size *string***)** ⇒ *number* operation

The **size** operation returns the number of characters in the string.

```
(size "A string with a \" in it")     ⇒ 23
```

(concat *string string* …**)** ⇒ *string* operation

7.3. Characters

Characters are objects that represent printed characters such as letters and digits. They are written using the notation `#\`⟨*character*⟩ or `#\`⟨*character escape*⟩. Characters are constants and hence evaluate to themselves. The character escapes are discussed in section **3.2** and include:

`#\\n #\\r #\\t #\\f #\\x` and `#\\\`

The escapes are used to write certain non-graphic characters and to write the backslash character.

(-> Char) ⇒ *operation* operation

The prototypical **Char** responds to the operation **->** with the operation **->char** that may be used with other kinds of actors (in particular **Fixnums**) to produce a character. The following comparison operations are defined on characters:

```
(= c1 c2)          (< c1 c2)          (> c1 c2)
(!= c1 c2)         (<= c1 c2)         (>= c1 c2)
```

The ordering is the conventional ASCII character set ordering. For example:

```
(< #\G #\g)            ⇒ #t
(< #\f #\G)            ⇒ #f
(< #\4 #\A)            ⇒ #t
```

(->fixnum *character***)** ⇒ *number* operation

Characters respond to the operation **->fixnum** to convert themselves to the equivalent number:

```
(->fixnum #\A)         ⇒ 65
```

The coercion from characters to numbers preserves the ordering of characters.

7.4. Numbers

Rosette 1.0 provides **Fixnums** and **Floats**. Future releases will provide **Bignums**.

(-> Fixnum) ⇒ *operation* operation

(-> Float) ⇒ *operation* operation

The numbers answer the **->** operation with the operations **->fixnum** and **->float** that may be used to convert other kinds of entities to numbers. For example, see sections **7.3** and **7.5**. The following comparison predicates are defined on numbers:

```
(= n1 n2)          (< n1 n2)          (> n1 n2)
(!= n1 n2)         (<= n1 n2)         (>= n1 n2)
```

7. Atomic Objects, Strings, and Tuples

This section discusses the operations on the atomic objects which include special constants, booleans, characters, numbers, and symbols. Strings and tuples are also discussed.

7.1. Special constants

The constant **`#niv`** is used as a value when there is no other appropriate value. For example, a slot of an actor might be initialized to **`#niv`** to indicate that no meaningful value is available. The operation **`niv?`** returns **`#t`** if it is sent to **`#niv`** and returns **`#f`** otherwise. The constant **`#absent`** is used to indicate that a key has no binding in some context. It is returned by such operations as **`lookup`**. The operation **`absent?`** returns **`#t`** if sent to **`#absent`** and **`#f`** otherwise. The constant **`#eof`** is the result of a **`read`** operation on an input stream when the end of the stream has been reached (see section **11**). The operation **`eof?`** returns **`#t`** if sent to **`#eof`** and **`#f`** otherwise.

7.2. Booleans

The boolean constants are **`#t`** and **`#f`**. As with all constants, they evaluate to themselves.

`(and` ⟨*expr*⟩ … **`)`** ⇒ *boolean* — derived syntax

The expressions are evaluated in text order. An expression that evaluates to **`#f`** terminates the construct, the result of the expression is **`#f`**, and the evaluation of the remaining expressions is not completed. If all expressions evaluate to **`#t`**, then the result of the expression is **`#t`**.

```
(and (= 2 2) (> 2 1))              ⇒ #t
(and (!= y 0) (= (/ x y) 4))       ⇒ #f  if y is 0
(and)                              ⇒ #t
```

`(or` ⟨*expr*⟩ … **`)`** ⇒ *boolean* — derived syntax

The expressions are evaluated in text order. An expression that evaluates to **`#t`** terminates the construct, the result of the expression is **`#t`**, and the evaluation of the remaining expressions is not completed. If all expressions evaluate to **`#f`**, then the result of the expression is **`#f`**.

```
(or (= 2 2) (> 2 1))                    ⇒ #t
(or (= 2 3) (< 1 2) (= (/ 3 0) 1))      ⇒ #t
(or)                                    ⇒ #f
```

`(not` ⟨*expr*⟩**`)`** ⇒ *boolean* — derived syntax

The derived expression **`not`** expects a boolean value and returns the negation of that value:

```
(not #t)                                ⇒ #f
(not #f)                                ⇒ #t
```

```
(deposit my-account 100.01) ⇒ ['deposited 100.01]
(withdraw my-account 78.87) ⇒ ['withdrew 78.87]
```

Now a **SafeAccount** can be defined that provides overdraft protection through a second account:

```
(defActor SafeAccount
  (extends& BankAccount)
  (slots& backup BankAccount)

  (method (withdraw amount)
    (cond ((<= amount balance)
           (update balance (- balance amount))
           ['withdrew amount])
          ((> amount balance)
           (let [[[status value] (withdraw backup (- amount balance))]]
             (if (same? status 'withdrew)
                 (block (update balance 0)
                        ['withdrew amount])
                 (block (update)
                        ['overdraft value])))))) )
```

The above definition extends **BankAccount** with a new slot **backup** to hold the secondary account, and overrides the method for **withdraw** so that the **backup** account is accessed if there are not enough funds in the **SafeAccount**. The default value for the **backup** is the prototypical empty **BankAccount**. A new **SafeAccount** may be created as follows:

```
(new SafeAccount 1000.00 (new BankAccount 2500.00))
```

It might be nice to be able to retrieve the current **balance** from the account. This behavior can be added by:

```
(defOprn enquire)

(defPure BankAccount (enquire) balance)
```

and of course the method for **enquire** will be inherited by **SafeAccounts** as well. Occasionally, it may be useful to add several new slots to a shared behavior object at once. This is supported by the following form.

(defSlots ⟨*expr*⟩ **(sbo&** ⟨*expr*⟩**')** ⟨*form*⟩ ...**)** derived syntax

The ⟨*expr*⟩ will typically evaluate to a prototype and the optional **(sbo&** ⟨*expr*⟩**')** clause will be omitted. In this case, the new slots are added to the object that results from **(sbo** ⟨*expr*⟩**)**. The ⟨*form*⟩s are as described above for **defActor**.

6.3. Defining actors and objects

(defActor ⟨*id*⟩ **(extends&** ⟨*expr*⟩**)** **(slots&** ⟨*id*⟩$_1$ ⟨*expr*⟩$_1$ …**)** ⟨*form*⟩ …**)** derived syntax

This form is the principal means of introducing new kinds of actors into the environment. The ⟨*id*⟩ is bound to the prototypical actor in the `Global` environment, and will also be returned in response to the **`kind`** operation. If the optional **(extends&** ⟨*expr*⟩**)** is supplied, then the new prototype will have at least the slots defined by the entity that results from ⟨*expr*⟩; otherwise, the prototypical empty object is extended which has **`Top`** as its shared behavior object. The clause **(slots&** ⟨*id*⟩$_1$ ⟨*expr*⟩$_1$ …**)** is optional and if present serves to define slots and default values in addition to those in the entity being extended. The ⟨*form*⟩s, if present provide initial bindings for methods, procedures, and values in the shared behavior object that is created for the new prototype. The ⟨*form*⟩s may be any of the following:

(method	**(**⟨*oprn*⟩ ⟨*pattern*⟩**)**	⟨*body*⟩**)**	derived syntax
(pure	**(**⟨*oprn*⟩ ⟨*pattern*⟩**)**	⟨*body*⟩**)**	derived syntax
(local	**(**⟨*id*⟩ ⟨*pattern*⟩**)**	⟨*body*⟩**)**	derived syntax
(proc	**(**⟨*id*⟩ ⟨*pattern*⟩**)**	⟨*body*⟩**)**	derived syntax
(value	⟨*id*⟩	⟨*expr*⟩**)**	derived syntax
(slot	⟨*expr*⟩	⟨*expr*⟩′**)**	derived syntax

The first four forms correspond to the defining forms of the previous section. The **`value`** form simply adds a slot with ⟨*id*⟩ bound to the value of the ⟨*expr*⟩, and **`slot`** permits an arbitrary value to be the key determined as the value of ⟨*expr*⟩ to be bound to the value of ⟨*expr*⟩′.

A simple bank-account actor may be defined as follows:

```
(defOprn withdraw)
(defOprn deposit)

(defActor BankAccount (slots& balance 0)
  (method (withdraw amount)
    (cond ((<= amount balance)
           (update balance (- balance amount))
           ['withdrew amount])
          ((> amount balance)
           (update)
           ['overdraft (- amount balance)])))
  (method (deposit amount)
    (update balance (+ balance amount))
    ['deposited amount]))
```

This defines two operations to be used to manipulate bank accounts and defines the prototypical bank account and an associated shared behavior object in which the methods for **`withdraw`** and **`deposit`** are stored. The prototypical bank account has a **`balance`** of `0`. A new **`BankAccount`** may be created by:

```
(define my-account (new BankAccount 1000.00))
```

Requests may be issued to the new account by using the operations that were defined above:

(defMethod ⟨*expr*⟩ **(**⟨*oprn*⟩ ⟨*pattern*⟩**)** ⟨*body*⟩**)** ⇒ ⟨*oprn*⟩ derived syntax
(defPure ⟨*expr*⟩ **(**⟨*oprn*⟩ ⟨*pattern*⟩**)** ⟨*body*⟩**)** ⇒ ⟨*oprn*⟩ derived syntax
(defLocal ⟨*expr*⟩ **(**⟨*id*⟩ ⟨*pattern*⟩**)** ⟨*body*⟩**)** ⇒ **'**⟨*id*⟩ derived syntax

These three forms are used to add method slots to the shared behavior object associated with an entity. The entity is determined as the result of evaluating the ⟨*expr*⟩. The methods are compiled with respect to this entity and then bound to either the ⟨*oprn*⟩ or ⟨*id*⟩ in the object identified by **(sbo** ⟨*expr*⟩**)**. The **defMethod** form corresponds to the **method** expression above, and the **defPure** to **pure** expressions. In these two forms, ⟨*oprn*⟩ should be an expression that evaluates to an **Operation**. These two forms are conventionally used to introduce methods that correspond to messages that are sent by clients of an entity. The form **defLocal** binds the resulting method object to the identifier ⟨*id*⟩. This form is conventionally used to introduce methods that are used locally by an actor or object and are not available to their clients. Local methods are principally a device to promote modularity and reuse within the definition of an actor or object.

(defOprn ⟨*id*⟩ **& (sbo&** ⟨*expr*⟩**)** ⟨*expr*⟩**')** ⇒ **'**⟨*id*⟩ derived syntax
(defOprn (⟨*id*⟩ ⟨*pattern*⟩**)** ⟨*body*⟩ **& (sbo&** ⟨*expr*⟩**) pure&)** ⇒ **'**⟨*id*⟩ derived syntax
(defSync ⟨*id*⟩ **& (sbo&** ⟨*expr*⟩**)** ⟨*expr*⟩**')** ⇒ **'**⟨*id*⟩ derived syntax
(defSync (⟨*id*⟩ ⟨*pattern*⟩**)** ⟨*body*⟩ **& (sbo&** ⟨*expr*⟩**) pure&)** ⇒ **'**⟨*id*⟩ derived syntax

These forms introduce new operations into the environment and allow for the initial binding of methods or other values to the new operation. Often these initial bindings represent the "default" behavior or value of the operation and are placed in the **Top** environment. This is accomplished by simply omitting the **(sbo&** ⟨*expr*⟩**)** clause. The forms with just an ⟨*id*⟩ and no ⟨*pattern*⟩ define a new operation without any default or with default as determined by the result of evaluating ⟨*expr*⟩**'**. The other two forms provide for the association of methods, and follow the same basic syntax of **defMethod**. The optional flag **pure&** may be used to indicate that the method is to be wrapped with an **(update)**.

```
(defOprn foo)                                  ⇒ 'foo
(defOprn kind '?)                              ⇒ 'kind
(defSync (examine)
  (examine! (meta (self)) (self) stdout)       ⇒ 'examine
```

The first example simply creates a new **Oprn** and binds it to **'foo** in the **Global** environment. The second example creates a new operation bound to **'kind** in the **Global** environment and binds the new operation to the symbol **'?** in the **Top** environment. The last example creates a **SyncOprn** and binds the new operation to the following method in the **Top** environment:

```
(method [] (examine! (meta (self)) (self) stdout))
```

The **defOprn** and **defSync** forms check to see whether the ⟨*id*⟩ is already bound to an operation and if so do not actually create a new operation but simply retain the current one, possibly adding a new binding for the operation.

6.2. Procedures, Methods, and Operations

(proc ⟨*pattern*⟩ ⟨*body*⟩**)** special form

The **proc** expression evaluates to (i.e., creates) an anonymous procedure object. When a **proc** is sent a message that matches ⟨*pattern*⟩, it evaluates the ⟨*body*⟩ of the **proc**. The ⟨*body*⟩ is evaluated in an environment that is extended with the bindings resulting from matching the ⟨*pattern*⟩ with the received message. The environment in effect at the point that the **proc** expression is evaluated is stored in a slot in the resulting procedure object. Procedure objects are always ready to receive a message and behave like the procedures that result from evaluating a **lambda** expression in Scheme. The ⟨*body*⟩ is an implicit block construct.

```
(proc [x] (+ x x))                      ⇒ {Proc instance}
((proc [x] (+ x x)) 4)                  ⇒ 8
(define twice                           ⇒ 'twice
  (proc [n] (+ n n))
(twice 4)                               ⇒ 8
```

(defProc (⟨*id*⟩ ⟨*pattern*⟩**) (sbo&** ⟨*expr*⟩**)** ⟨*body*⟩**)** ⇒ **'**⟨*id*⟩ derived syntax

The **defProc** form defines ⟨*id*⟩ to be a procedure object that accepts messages according to ⟨*pattern*⟩. The ⟨*body*⟩ is an implicit block as with the **proc** expression. If the optional **sbo&** is omitted then the definition is made in the **Global** environment; otherwise, the procedure object is bound to ⟨*id*⟩ in the object resulting from ⟨*expr*⟩. The familiar factorial may be expressed as follows:

```
(defProc (fact n)
  (if (< n 2) 1 (* (fact (- n 1)) n)))  ⇒ 'fact
```

Procedures defined using **defProc** may be recursive without any extra effort.

(method ⟨*pattern*⟩ ⟨*body*⟩**)** special form
(pure ⟨*pattern*⟩ ⟨*body*⟩**)** derived syntax

The **method** expression evaluates to an anonymous method object. The **pure** expression simply expands to a **method** expression which has the form **(update)** added to unlock the entity immediately. This form is provided since it is quite common to have methods that are essentially functional. In any case the ⟨*body*⟩ is evaluated in an environment that is extended with the bindings resulting from matching the ⟨*pattern*⟩ with the received message. The environment that is extended is that representing the entity that received the message that invoked the method. The body of a **method** (or **pure**) is an implicit block.

6. Definitions

This section discusses defining forms. The **`proc`**, **`method`**, and **`pure`** expressions and related defining forms are used to create objects that provide executable behavior. The forms **`defActor`** and **`defObject`** are used to create new families of actors and objects. **`defOprn`** and **`defSync`** define new operations.

6.1. Message patterns

In the following, ⟨*pattern*⟩ is the form of messages that will be accepted by procedures or methods, and specifies variables that will be "bound" to portions of the received messages. The ⟨*pattern*⟩ may take one of the following forms (these apply to the following definition forms as well):

- **`[`**⟨*id*⟩ …**`]`**: Indicates that the actor accepts messages (represented as tuples) with a fixed number (0 or more) of elements; when the actor receives a message, the elements of the message are bound in turn to the ⟨*id*⟩s in the ⟨*pattern*⟩. Note that the list of ⟨*id*⟩ may be empty in which case the actor is "triggered" by receipt of an empty message.

```
((proc [] 4))                          ⇒ 4
```

- **`[`**⟨*id*⟩ … **`&`** ⟨*id*⟩**`]`**: Indicates that the actor accepts messages with at least as many elements as there are ⟨*id*⟩s in the pattern preceding the **`&`**. The ⟨*id*⟩ following the **`&`** is bound to a message (possibly empty) containing the rest of the elements.

```
((proc [a b & r] r) 1 2 3 4 5)         ⇒ [3 4 5]
```

- **`[`**⟨*pattern*⟩ …**`]`**: Indicates that the actor accepts messages that have a fixed number of elements which may in turn be messages. The bindings are established by recursively "destructuring" the received message according to the patterns.

```
((proc [[a & r] [b & s]] [a b]) [1 2 3] [4 5 6])⇒ [1 4]
```

- **`[`**⟨*pattern*⟩ … **`&`** ⟨*id*⟩**`]`**: Indicates that the actor accepts messages with at least as many elements as there are ⟨*pattern*⟩s preceding the **`&`**, with the remainder of the elements (possibly none) bound to the ⟨*id*⟩ following the **`&`**.

```
((proc [[a b] & r] [a b r])
 [1 2] [3 4] [5 6])                    ⇒ [1 2 [[3 4] [5 6]]]
```

The identifiers occurring in a pattern must all be distinct, and a **`formals-mismatch`** error is raised if a message is sent to an entity that does not match its pattern.

5.5. Miscellaneous forms

This section covers the following forms: **`free`**, **`goto`**, **`label`**, **`set!`**, and **`void`**.

`(free [`⟨*id*⟩ …**`]`** ⟨*body*⟩**`)`** ⇒ *result of* ⟨*body*⟩ special form

The **`free`** form informs the compiler that the ⟨*id*⟩ are to be treated as free in ⟨*body*⟩; otherwise, the compiler will issue warnings that the ⟨*id*⟩ do not have a compile-time binding. Sometimes this is exactly what is desired. For example, if one of the identifiers names an inherited slot holding a value shared across all descendants of some entity or names a "local" method that is inherited. In other cases, the warning indicates that an error has been made in defining the method or procedure. Thus, it is good form to use **`free`** to document the intent that the ⟨*id*⟩s are to be looked up via inheritance.

The next three forms are intended mainly for use in constructing the internals of forms such as the iteration constructs in the previous section and other "low-level" system code. They are mentioned here for completeness and may well be removed.

`(goto` ⟨*id*⟩**`)`** ⇒ *no result* special form
`(label` ⟨*id*⟩ ⟨*body*⟩**`)`** ⇒ *result of* ⟨*body*⟩ special form
`(set!` ⟨*id*⟩ ⟨*expr*⟩**`)`** ⇒ *any* special form

The **`goto`** and **`label`** forms provide a primitive looping construct. Essentially, the **`goto`** can only branch backward, forward branching is accomplished via **`if`**. The **`set!`** is a primitive assignment that permits altering bindings in the lexical environment of a procedure or method.

`(void` ⟨*body*⟩**`)`** ⇒ **`#niv`** derived syntax

The **`void`** form is provided as a convenient notation for indicating that a form is not really intended to yield a result. The compiler will issue a warning in those contexts that may be required to produce a result, such as the body of a procedure or method. The compiler is conservative since it can be quite difficult to track down errors caused by a procedure or method not returning a result when one was needed. The usual symptom of such an error is that activity simply ceases much as when an actor or object is not unlocked. The **`void`** form expands as follows:

`(void` ⟨*body*⟩**`)`** *expands to* **`(block #niv`** ⟨*body*⟩**`)`**

5.4. Iteration

(iterate ⟨*id*⟩ **[[**⟨*id*⟩$_1$ ⟨*expr*⟩$_1$**]** … **]** ⟨*body*⟩**)** ⇒ *any* — derived syntax

iterate is one of the forms used to introduce loops in procedures and methods. The ⟨*id*⟩ becomes the name for an implicit procedure that may be executed from within the ⟨*body*⟩ in order to repeat the ⟨*body*⟩. The ⟨*expr*⟩$_i$ are evaluated concurrently and bound to the ⟨*id*⟩$_i$ initially and the ⟨*body*⟩ is evaluated in an environment that is extended with these bindings. If the ⟨*body*⟩ needs to repeat the loop then a request of the form:

(⟨*id*⟩ ⟨*expr*⟩ …**)**

is made. This causes the ⟨*body*⟩ to be evaluated again with the bindings determined by the ⟨*expr*⟩. The following example computes the factorial of **12**:

```
(iterate loop [[n 12] [r 1]]
  (if (< n 2)
      r
      (loop (dec n) (* r n))))              ⇒ 479001600
```

The result of the **iterate** is the final result of the ⟨*body*⟩

(do [[⟨*id*⟩$_1$ ⟨*init*⟩$_1$ ⟨*step*⟩$_1$**]** … **] [(**⟨*cond*⟩$_1$ ⟨*expr*⟩$_1$**)** … **]** ⟨*body*⟩**)** ⇒ *any* — derived syntax
(do* [[⟨*id*⟩$_1$ ⟨*init*⟩$_1$ ⟨*step*⟩$_1$**]** … **] [(**⟨*cond*⟩$_1$ ⟨*expr*⟩$_1$**)** … **]** ⟨*body*⟩**)** ⇒ *any* — derived syntax

These two forms evaluate the ⟨*init*⟩$_i$ concurrently and bind the results to the ⟨*id*⟩$_i$. The ⟨*cond*⟩$_i$ are then evaluated in an environment extended with the bindings and if none are **#t**, then the ⟨*body*⟩ is evaluated in the same environment, the ⟨*step*⟩$_i$ are evaluated and rebound to the ⟨*id*⟩$_i$, and the process repeats. The two forms differ in how the ⟨*step*⟩$_i$ are computed. In the case of **do** the ⟨*step*⟩$_i$ are computed concurrently and for the **do*** they are evaluated in text-order and the subsequent ⟨*step*⟩s may depend on the bindings established by preceding ⟨*step*⟩s. For example,

```
(do [[last #niv (new List n last)]
     [n    1    (inc n)]]
  [((= n N)  (new List n last))])
```

is a simple loop that builds **N List** actors such that the head of the list holds the value **N** and successive **List** actors have decreasing values down to **1**. The variable **last** is initially **#niv** and **n** is initially **1**. The termination condition is **(= n N)** at which point a final **List** actor is created as the result of the loop. On successive steps through the loop **last** is bound to a new **List** actor with the current value of **n** and linked to the current value of **last**, also **n** is incremented. In this example the ⟨*body*⟩ is empty which is not unusual since often all of the computation can be performed in the ⟨*step*⟩$_i$.

```
(cond ((<= amount balance)
        (update balance (- balance amount))
        ['withdrew amount])
       ((> amount balance)
        (update)
        ['overdraft (- amount balance)]))
```

5.3. Binding constructs

The binding constructs **`let`**, **`let*`** and **`letrec`** give Rosette a block structure, as in Common Lisp or Scheme. These constructs differ in the regions they establish for variable bindings.

`(let [[`⟨*id or pattern*⟩ ⟨*expr*⟩**`]`** … **`]`** ⟨*body*⟩**`)`** special form

The ⟨*expr*⟩s are evaluated concurrently, the results are then bound according to the corresponding ⟨*id or pattern*⟩s. The ⟨*body*⟩ is then executed in an environment extended with the indicated bindings. The result of the ⟨*body*⟩ is returned to the continuation that was in effect at the point where the **`let`** occurred. A ⟨*pattern*⟩ is used in a binding to allow the ⟨*expr*⟩ to return multiple values. (for more on the syntax of ⟨*pattern*⟩s, see section **6.1**.) The ⟨*body*⟩ is an implicit block.

```
(let [[x 2] [y 3]] (* x y))            ⇒ 6
```

`(let* [[`⟨*id or pattern*⟩ ⟨*expr*⟩**`]`** … **`]`** ⟨*body*⟩**`)`** derived syntax

The **`let*`** form expands to nested **`lets`** one for each `[`⟨*id or pattern*⟩ ⟨*expr*⟩`]`.

```
(let* [[x 2]         [y (+ x 1)]
       [x (+ y 4)] [z (+ x 9)]]
  (+ x y z)))                          ⇒ 26
```

`(letrec [[`⟨*id*⟩ ⟨*expr*⟩**`]`** … **`]`** ⟨*body*⟩**`)`** special form

The ⟨*id*⟩s are initially bound to “dummy” actors that will **`become`** the results of the ⟨*expr*⟩s. The environment in effect at the point of occurrence of the **`letrec`** is extended with these initial bindings. This extended environment is then used to evaluate the ⟨*expr*⟩s concurrently. As each ⟨*expr*⟩ produces a result the corresponding dummy actor **`become`**s that result. The body is then evaluated in this extended environment with the continuation that was in effect for the entire **`letrec`**. The ⟨*body*⟩ is an implicit block construct.

```
(letrec [[even? (proc [n] (if (= n 0) #t (odd? (- n 1))))]
         [odd?  (proc [n] (if (= n 0) #f (even? (- n 1))))]]
  (even? 88))                                      ⇒ #t
```

In this example, the two **`proc`** expressions (see section **6.2**) are evaluated in an environment in which both **`even?`** and **`odd?`** are given fresh bindings. Thus, the two **`proc`** expressions will be closed over an environment in which each is defined and visible to the other.

(seq $expr_1$ … $expr_n$**)** $\Rightarrow$ $result_n$ — special form

This form evaluates a sequence of expressions in “text order” and has as a result the result of the last expression in the **seq** form. This form is included in Rosette along with **set!**, **label**, and **goto** in order to allow the expression in Rosette of methods that would otherwise have to be primitively implemented. These include most of the I/O system and the iteration constructs provided in Rosette as syntax extensions. It should rarely be the case that these forms are used other than in such contexts. Each of the $expr_i$ must return a result, since this is the only way in the Actor model that causal dependencies can be expressed. Thus, if a send is used in a **seq** form, it should be enclosed in a **void** form (see below).

5.2. Conditionals

(if $\langle expr\rangle_1$ $\langle expr\rangle_2$**)** — special form
(if $\langle expr\rangle_1$ $\langle expr\rangle_2$ $\langle expr\rangle_3$**)** — special form

These two forms provide the conventional if-then and if-then-else control constructs in which the *then* and *else* parts are **not** evaluated unless the condition warrants it. If $\langle expr\rangle_1$ evaluates to **#t**, then the first form returns the result of evaluating $\langle expr\rangle_2$, and returns **#niv** if the condition evaluates to **#f**. The second form behaves as the first except that $\langle expr\rangle_3$ is evaluated for the case of **#f**. If $\langle expr\rangle_1$ does not evaluate to a boolean, it is treated as **#t**.

```
(if #t 1)                    ⇒ 1
(if #f 1)                    ⇒ #niv
(if (= y 0) 0 (% x y))       ⇒ 0 or (% x y)
```

(cond ($\langle test\rangle_1$ $\langle body\rangle_1$**) (**$\langle test\rangle_2$ $\langle body\rangle_2$**)** …**)** — derived syntax
(cond ($\langle test\rangle_1$ $\langle body\rangle_1$**) (**$\langle test\rangle_2$ $\langle body\rangle_2$**)** … **(else** $\langle body\rangle_e$**))** — derived syntax

The **cond** packages a familiar use of **if** in which there are several (usually mutually exclusive) conditions to be tested for and an action taken on some true condition. The $\langle test\rangle_i$ are evaluated concurrently and the $\langle body\rangle_i$ corresponding to a true $\langle test\rangle_i$ is evaluated and the result returned to the continuation of the entire **cond**; if there is more than one true $\langle test\rangle_i$, then one $\langle body\rangle_i$ is chosen non-deterministically for evaluation. If all of the $\langle test\rangle_i$ are evaluated and none are **#t**, then if an **else** clause has been specified, the $\langle body\rangle_e$ is evaluated; otherwise the result of the **cond** is **#niv**. The **cond** will not terminate if not all of the $\langle test\rangle_i$ complete, and among the ones that do complete none are **#t**. Each of the $\langle body\rangle_i$ is an implicit block construct. It can always be ensured that the result of a **cond** is deterministic and terminating by writing each of the $\langle test\rangle_i$ so that for all contexts, exactly one evaluates to **#t**, as in the following:

5. Compound constructs

The compound constructs are built using the simple expressions and commands of section **4** and recursive use of the compound constructs. The compound constructs are presented in terms of the categories: blocks, conditionals, and binding constructs.

5.1. Blocks

`(block` ⟨*expr*⟩ ⟨*expr*⟩ … **`)`** basic syntax

A block expression packages a set of expressions that are to be evaluated concurrently. Again, "concurrently" means that the expressions may be executed in any order and the semantics of the block expression should not depend on any particular order. The continuation in effect at the point the block expression is encountered becomes the continuation for each of the expressions in the block. This is a crucial difference between the **`block`** and the processing of the components of communication constructs. The communication constructs build new continuation actors to receive results and package them in a message structure. The **`block`** does not build any new continuations; it forwards its continuation to its component expressions.

If an expression is a send expression, then the continuation is not used. If the expression is a request expression then the continuation is propagated in the request to be used as the target of the result message. For other expressions, the situation is similar to that of a request. For example, a constant expression simply returns itself to the continuation at the point the constant is encountered.

Since all of the expressions in a block receive the same continuation, it is possible to directly represent a "race" among competing expressions, and the first result to be delivered to the actor representing the waiting context becomes the result of the block. Later results, if any, are ignored. The following example illustrates the non-determinism inherent in the block construct:

`(block 1 2 3 4)` ⇒ **`1`** *or* **`2`** *or* **`3`** *or* **`4`**

The behavior above is a consequence of the "natural" semantics of a block. More conventional uses of the block construct have at most one expression that will yield a result. All other expressions will be sends, or other compound expressions that do not yield results.

```
(block
  (update balance (+ balance amount))
  ['credited amount])
```

In the example above, updating the **`balance`** does not produce any result, while two values (**`'credited`** and **`amount`**) are sent to the continuation waiting for the result of the block. If a block is encountered that may produce more than one result then a warning is issued to that effect.

(become *prototype* **&** *initialization_arguments***)** derived syntax
(become! *prototype***)** ⇒ **(self)** primitive

The **become** form causes the actor executing it to become an instance of *prototype* after executing **init** with the *initialization_arguments*. **become** permits an implementation to construct the new instance in place of the actor that executes the **become**, thus there need not be any forwarding overhead as with the more general **become** primitive defined in [Agha 1986]. The primitive **become!** causes the current **(self)** to become a clone of the *prototype*. The result of the primitive is a reference to **(self)**. The **become** syntax expands as follows:

(become *prototype* **&** *initialization_arguments***)**

expands-to

(send init (become! *prototype***) &** *initialization_arguments***)**

analogously to **new**. **become** may be viewed as providing a generalized form of "dynamic inheritance" in which not only the parent of an entity is changed but its structure as well. This is a useful technique for decomposing the implementation of a data abstraction along the lines of its abstract specification. For example, queues are often specified in terms of the empty queue and the non-empty queues. In this context, it would be appropriate to define a family empty queues and non-empty queues in the implementation. The method for enqueueing an item on an empty queue might be written as follows:

```
(defMethod Queue (enq item) (become QHead item))
```

In this case, **Queue** is the prototypical empty queue and **QHead** the prototype for non-empty queues. An empty queue instance then changes its behavior to that of a non-empty queue when it receives an **enq** request.

entity begins processing a message it is “locked” until one of these constructs is executed, at which point the actor is “unlocked” and may begin processing another message. It is worth emphasizing that until an actor is unlocked it can not process any further messages. This is a common source of bugs in Rosette programs and manifests itself by the mysterious ceasing of computation with no result. During the processing of a message, exactly one of these constructs should be executed. The effect of executing more than one is undefined, but may give rise to:

```
***warning: EmptyMbox::nextMsg invoked
```

In any event, it is far more common to fail to unlock an entity than it is to issue multiple state changes.

(update ⟨*id*⟩ ⟨*expr*⟩ …**)** — derived syntax
(update! <*keyExpr*> ⟨*expr*⟩ …**)** ⇒ **(self)** — primitive

The **update** form is used to change one or more slots (acquaintances) of the entity executing the **update**. As an example, if **x** has the value **4** and **y** has the value **2**, and the following two forms are executed together (see section **5.1**), the results **4** and **2** will be returned. For the next message processed, the values of **x** and **y** will be **3** and **8** respectively.

```
(update x (+ y 1) y (* x 2))
[x y]
```

The update expression may be empty (i.e., just **(update)**), which simply causes the entity to become unlocked with no changes in its state. The **update** construct is derived using the primitive **update!**. **update** is used when all of the keys are identifiers (which is the usual case); while **update!** permits the bindings for arbitrary keys to be modified.

(next *enabled-set* ⟨*id*⟩ ⟨*expr*⟩ …**)** — derived syntax
(next! *enabled-set* <*keyExpr*> ⟨*expr*⟩ …**)** ⇒ **(self)** — primitive

These two forms are generalizations of **update** which permit the entity to change the *enabled-set* that will be used to accept messages when it becomes unlocked. In Rosette 1.0, an *enabled-set* is simply a **Tuple** of message prefix patterns, which are in turn represented as **Tuples**. For example, an empty queue might reasonably want to restrict the messages that it will accept to only **enq** messages. In this case, the prefix of the message has the form **[enq]** and the *enabled-set* would simply be **[[enq]]**:

```
(next [[enq]])
```

The above would unlock the queue entity and permit only **enq** messages to be received. For more information on enabled-sets see [Tomlinson 1989a].

is analogous to a conventional procedure or function call. A request to the target will yield a result which is sent to the continuation of the request. The textual position of a request expression is to be viewed as being replaced by the result of the request. The notion of continuation is that of the actions that are to occur after the result of a request has been received. A continuation is represented as an actor that waits on the result and then initiates further actions. The mail-address of the actor representing the continuation is sent (implicitly) in the message. When a request is received by the target actor it is evaluated in an environment that includes the continuation received in the request. The result of the evaluation of the request is sent to this continuation.

```
(+ 3 4)                                  ⇒ 7
(+ (* 3 4) (* 4 5))                      ⇒ 32
```

Since messages are represented via `Tuples` it will sometimes be the case that the message or a part of the message will already be available and can be sent directly to the target. The form "`&` ⟨*expr*⟩" is used to include the elements of a tuple in the message to be sent to some actor. Suppose that `a` is bound to the tuple `[3 4]`, then the elements of this tuple may be sent in a request to "`+`" as follows:

```
(+ & a)                                  ⇒ 7
```

The construct `(send` ⟨*expr*⟩ ⟨*Clause*⟩`)` sends the message resulting from ⟨*Clause*⟩ to the target actor resulting from ⟨*expr*⟩ and does not produce a result. The send expression may be viewed as a command. That is, an action is initiated that is intended to cause some effects and not to produce a result. The message is sent asynchronously to the target actor. In the example below, a queue actor is sent a message to update the queue with a new entry. No result is to be produced, just the side effect of the update.

```
(send enq a-queue 3)                     ⇒ no result
```

In concurrent programs, it is often the case that multiple items of information are generated and to be sent to the continuation for distribution among several points of use. In Rosette this simply accomplished by returning a `Tuple` consisting of the multiple items. The binding constructs discussed later in section **5.3** make it particularly easy to destructure multiple values returned in this way.

4.5. Modifying behavior

The Actor model specifies that there are three basic capabilities of any actor: creating new actors, sending messages, and changing behavior. The first two capabilities have already been introduced in sections **4.3.1** and **4.4**. The third basic capability is provided by the constructs discussed in this section. They are used to change the behavior of an entity for the next message that it processes. They have no effect on the execution of the entity with respect to the current message being processed. When an

4.3.3. Built-in procedures

For each behavior to which an operation given above applies, there is a procedure that is used to perform the task. Many of these procedures are implemented primitively in the Rosette virtual machine and are available in the initial environment. Their names may be determined from the operation name and name of the most general kind of entity to which they apply. There are three exceptions to this rule: the names of primitive methods on **`Fixnums`** are prefixed with **`fx`**, for **`Floats`** the prefix is **`fl`**, and for **`Chars`** it is **`ch`**. The rule for non-coercer operations is to prefix the operation name with the kind name in lower case followed by a **–** (dash) if the operation name begins with an alphabetic character. For example, the primitive method corresponding to **`size`** on **`Strings`** is **`string-size`**. If the operation name begins with an extended alphabetic character, no **–** is used. For example, the procedure corresponding to **=** on **`Fixnums`** is **`fx=`**. The full name of a coercion procedure is obtained by using the above rule for regular operations on the name of a specific coercion operation. For example, **`->string`** as an operation on **`Symbols`** corresponds to a primitive method with the name **`symbol->string`**. It should be emphasized that this naming approach is purely a convention for choosing the identifiers in the initial environment and has no operational significance. (The reader should be warned that in the 1.0 release of Rosette the naming of primitive methods has not necessarily been rigorously followed.)

4.4. Communication constructs

(⟨*expr*⟩ ⟨*Clause*⟩**)** ⇒ *result* — basic syntax
(send ⟨*expr*⟩ ⟨*Clause*⟩**)** — special form

One of the fundamental capabilities of actors according to the Actor model is that of sending messages to other actors. The communication constructs are the Rosette notation for sending messages. The ⟨*expr*⟩ is an expression that evaluates to the entity which is the target of the communication. A ⟨*Clause*⟩ has one of the two forms:

⟨*expr*⟩ … or ⟨*expr*⟩ … **&** ⟨*expr*⟩

The ⟨*Clause*⟩ is a sequence of 0 or more expressions that are evaluated to determine the message that is to be sent to the target. The last portion of the ⟨*Clause*⟩ may be of the form "**&** ⟨*expr*⟩". This form is used to include a variable number of elements in the message. Both ⟨*expr*⟩ and ⟨*Clause*⟩ are evaluated concurrently. In practice, this will often mean that they are evaluated in some indeterminate order (possibly in parallel) and the semantics of a program can not depend upon any specific order.

The request expression **(**⟨*expr*⟩ ⟨*Clause*⟩**)** sends the message that results from evaluating ⟨*Clause*⟩ to the target entity that results from evaluating ⟨*expr*⟩ (usually the target is an **`Oprn`** or **`SyncOprn`**). It

(show-code new 'a) *displays*

```
litvec:
   0:   {MethodExpr}
   1:   {Template}
codevec:
   0:   extend 1
   3:   alloc 2
   6:   self 0, arg[0]
  10:   clone 1, arg[0]
  14:   xfer lex[0,0], arg[1]
  17:   xfer global[269], trgt
  21:   xmit/unwind/nxt 1
```

Briefly and referring to the example above for **source@**, the new method adds an environment contour to the receiver of the **new** message (e.g., **'a**) via **extend**, a new message to hold two arguments is built via **alloc 2**, the **self** primitive is executed leaving its result in the first argument position (**arg[0]**) of the new message, the **clone** primitive is executed on the result of **self** and its result is left in **arg[0]** and the initialization arguments (**args** above) are transferred to **arg[1]**, the binding for **init** in the **Global** environment is put in the **trgt** register of the virtual machine and finally the message is **xmit**ed after **unwind**ing the **args** into the message and control of the virtual machine is transferred to the **nxt** available piece of work. The result of the **init** operation will be returned to the context that invoked **new**.

(-> *any***)** ⇒ *operation* — operation

The **->** operation, when defined on an object, yields an operation that can be used to coerce other kinds of objects to its own kind. (See sections **7.4**, **7.5**, and **7.7** for examples.) The coercion protocol provides an interface that allows a program to perform coercions or conversions on arbitrary types of entities, and is an example of a higher-order use of operations. The coercion protocol is modeled after that of [Lang and Perlmutter 1986]. In the initial environment, specific coercion operations on various built-in objects are given names. The names are derived from the name of the generator object by prefixing **->** to the name of the generator object in lowercase. For example, the operation **(-> String)** is named **->string** in the initial environment. Thus, in the initial environment the following produce equivalent results:

((-> String) 23) ⇒ **"23"**

and

(->string 23) ⇒ **"23"**

This protocol is provided so that there is a uniform and extensible approach to type coercions.

the next object to be searched will be **`{Indexable SBO}`** and if that fails then **`{Expr SBO}`** and failing that the search continues with the **`(parent {Expr SBO})`**.

`(source@` *any***`)`** ⇒ *expr* — sync-operation
`(source@` any_1 any_2**`)`** ⇒ *expr* — sync-operation

The **`source@`** operation retrieves the source expression for *any* entity or for the binding of any_1 in any_2. If there is no binding for any_1 in any_2 then **`#absent`** is returned. If *any* is a Rosette procedure then its source is returned, otherwise the standard source for an object is returned. The standard response of an object when invoked is to return itself and the standard source reflects this. For example:

```
(source@ 1)      ⇒ '(pure [] (self))
(source@ new 1)  ⇒ '(method [& args] (init (clone (self)) & args)
```

`(formals@` *any***`)`** ⇒ *tupleExpr* — sync-operation
`(formals@` any_1 any_2**`)`** ⇒ *tupleExpr* — sync-operation

Similar to **`source@`**, the operation **`formals@`** will return the *tupleExpr* that represents the pattern of messages accepted by an entity. For example,

```
(formals@ 1) ⇒ '[]
(formals@ new 1) ⇒ '[& args]
```

This operation is particularly useful in determining the calling sequence expected by an operation on an object. If any_1 is not bound in any_2 then **`#absent`** is returned.

`(code@` *any***`)`** ⇒ *code* — sync-operation
`(code@` any_1 any_2**`)`** ⇒ *code* — sync-operation

The Rosette system compiles expressions to a byte code sequence in order to evaluate the expression. The **`code@`** operation returns this *code* object or **`#absent`** if any_1 is not bound in any_2. A related and perhaps more instructive operation is:

`(show-code` *any***`)`** ⇒ **`#niv`** — sync-operation
`(show-code` any_1 any_2**`)`** ⇒ **`#niv`** — sync-operation

This operation disassembles the *code* object that is returned by **`code@`** and displays it on **`stdout`**. For example, The **lookup** primitive will return the entity bound to any_1 in the entity returned by **(where** any_1 any_2**)**. Thus:

```
(lookup map '[a b]) ⇒ {Method instance}
```

yields the Rosette method object that implements the **map** operation on **TupleExprs**. If the slot identified by any_1 is not defined on any_2 then the result of **lookup** will be the constant **#absent**.

The following operations may be used to browse through the Rosette system. They may be used in conjunction with the above operations on the inheritance hierarchy to investigate the structure of the Rosette system. Eventually, graphical interfaces will be provided to make this process more convenient.

(kind *any***)** ⇒ *symbol* sync-operation

Every entity in the Rosette system answers to the **kind** operation with a symbol that names the prototype from which the object was cloned. Thus,

```
(kind 1)     ⇒ 'Fixnum
(kind 1.1)   ⇒ 'Float
(kind '[1])  ⇒ 'TupleExpr
(kind kind)  ⇒ 'SyncOprn
```

(describe *any***)** ⇒ **#niv** sync-operation
(describe *Ostream any***)** ⇒ **#niv** sync-operation

It is often very useful to obtain a simple description of an entity. For this purpose the **describe** operation is provided. For atoms, the result is simply to render the atom on either **stdout** or on the *Ostream* provided. For other kinds of entities the **kind** and the "role" of the entity in the system is given:

```
(describe 1)                displays 1
(describe '[1])             displays {TupleExpr instance}
(describe (parent '[1]))    displays {TupleExpr SBO}
```

(examine *any***)** ⇒ **#niv** sync-operation
(examine *Ostream any***)** ⇒ **#niv** sync-operation

The **examine** operation may be used with any entity to obtain a display of the structure of the entity. The result is rendered on either **stdout** or on the *Ostream* provided. Using the **examine** operation on the **(parent (sbo '[a b]))** from above gives the following:

```
{MIObject: Parent={MIObject SBO},
           Slots=['cpl [{Indexable SBO} {Expr SBO}]]}
```

This shows that an **MIObject** has a slot called **'cpl** (for class precedence list) which includes two shared behavior objects. Thus if a slot is not found in either a **TupleExpr** or its immediate SBO, then

(sbo 23) ⇒ **{Fixnum SBO}**

(parent *any***)** ⇒ *any* primitive

The **parent** primitive is applicable to any entity whatsoever in the Rosette environment and will return the next entity in the inheritance hierarchy. Rosette permits any entity to inherit from any other entity. For example, a point actor could inherit from another point actor. Thus, in general:

(sbo *any***) != (parent** *any***)**

Further, in Rosette a specialized family of objects, called **MIObjects** (for Multiple Inheritance Object) are used to provide the basic multiple-inheritance protocol (see section **8.5**). These objects manage the search among multiple parents and are not the direct repositories of inherited slots. Thus, the **parent** primitive will not necessarily yield the next object from which slots will be inherited by an object. In order to correctly identify the next object from which slots will be inherited the **parent-wrt** operation is provided.

(parent-wrt any_1 any_2**)** ⇒ *any* sync-operation

The **parent-wrt** synchronous operation returns the next object that will be searched after any_2 when starting at any_1. For example:

(parent-wrt '[a b] (sbo '[a b])) ⇒ **{Indexable SBO}**

but

(parent (sbo '[a b])) ⇒ **{MIObject instance}**

(where any_1 any_2**)** ⇒ *any* sync-operation

Related to **parent-wrt** is the **where** operation which returns the object "nearest" to any_2 in the inheritance hierarchy where the slot identified by any_1 will be located during inheritance search. If it is desired to find the object in which the method for the **map** operation on **TupleExprs** is located then the following request suffices:

(where map '[a b]) ⇒ **{Indexable SBO}**

That is, there is a slot identified by (the operation bound to the symbol) **map** in **{Indexable SBO}** when the inheritance search is started at **'[a b]**. This slot will contain a method for mapping a procedure over the elements of a **TupleExpr**. If the slot identified by any_1 is not defined on any_2 then the result of **where** will be the constant **#absent**.

(lookup any_1 any_2**)** ⇒ *any* primitive the result may not always be unique. For example, **(new [])** yields the unique empty tuple. The definition of **new** used by most objects is essentially:

(init (clone (self)) & *initialization_arguments***)**

The **new** operation is usually used to create a single instance of an entity and give it some arguments that the new instance may use to initialize itself. By convention the operation **newN** is provided on objects that implement a collection abstraction of some sort and for which it may be useful to create an instance with **N** elements initialized to some common value. Further, the operation **new*** is used conventionally to create an instance of a collection in which there are different initial values for the elements. For example:

(newN Tuple 4 'a) ⇒ **['a 'a 'a 'a]**

and

(new* TupleExpr 'a 'b 'c) ⇒ **'[a b c]**

Aspects of these operations that are specific to different kinds of entities are discussed in later sections.

(clone *any***)** ⇒ *a copy of any* primitive

The clone primitive usually produces a copy of its argument and is a basic creation primitive. In the cases: **#niv**, **#absent**, **#eof**, **Fixnum**, **Char**, **Symbol**, **EmptyMbox**, **LockedMbox**, **[]**, and **'[]**, the result produced by **clone** is the **same?** as its argument. In other cases, a distinct object is produced by "shallow" copying its argument. **Actors** are always cloned with a **LockedMbox** to ensure that entity has an opportunity to initialize itself before receiving any external messages. Initialization is accomplished via the **init** protocol. It should be noted that cloning immediately gives the new instance default values for its slots as determined by the values of the slots of the argument to clone.

(init *any initialization_argument …***)** ⇒ *any* sync-operation

The **init** synchronous operation serves to signal a (new) instance to initialize itself. Typically, the arguments provided are used to override some or all of the default values of the slots of the new instance. The **init** method must unlock the new instance so that it can receive messages and update its slots with the results of initialization (see section **3.5**).

The next group of operations can be used to traverse the inheritance hierarchy of the system, and are used in the implementation of the system itself.

(sbo *any***)** ⇒ *SharedBehaviorObject* sync-operation

The operation **sbo** can always be applied to any entity and will return the shared behavior object for its argument that is "nearest" in the inheritance hierarchy. For example:

deq	**describe!**	**display!**
empty?	**enq**	**examine!**
exp	**expand**	**expander**
expt	**failed?**	**floor**
flush!	**form**	**head**
iota	**lg**	**lgf**
log10	**log**	**logand**
lognot	**logor**	**logxor**
map	**max**	**mdiv**
min	**nth**	**pop**
print!	**push**	**rcons**
read!	**role**	**sbo?**
set-nth	**size**	**split**
state	**sub-obj**	**tail**
top	**walk**	**xchg**

The following identifiers are bound to synchronous operations in the initial Rosette environment:

add*	**body@**	**code@**
describe	**display**	**examine**
formals@	**formals-mismatch**	**gen**
init	**kind**	**locked?**
messages	**missing-method**	**new***
new	**newN**	**parent-wrt**
print	**print\n**	**runtime-error**
sbo	**show-code**	**source@**
vm-error	**where**	

Some of the above are generic in that they apply to virtually any entity. Many of these will be discussed here, and the others will be left to later sections devoted to the objects to which they apply. Some of the primitive operations are also discussed here.

(same? *any*$_1$ *any*$_2$**)** $\Rightarrow$ *boolean* primitive

A fundamental operation is **same?** which tests whether two entities have the same mail-address. This operation is applicable to any two actors, and returns **#t** or **#f** accordingly. It should be noted that while **same?** works to test equality on **Fixnums** it does not serve as equality on **Floats**. In these cases, the **=** operation or the associated primitives (**fx=** and **fl=** should be used).

The following operations define the basic creation protocols for entities in the Rosette system.

(new *prototype initialization_argument ...***)** $\Rightarrow$ *a new entity* sync-operation

In the Actor model, one of the fundamental capabilities that actors have is that of creating new actors. In Rosette, this is usually accomplished via the **new** operation. This operation is generally applicable to any object. For example, the prototype **Oprn** may be used to create new operations via the request:

(new Oprn *<identification>***)**

Depending on the prototype, additional arguments may be supplied that serve to initialize the instance. There is nothing special about prototypes, **new** copies of any entity can be requested; however,

4.3.2. Operations and Primitives

An operation is an **`Actor`** that determines the protocol for finding a method to perform some request or command. The concept of operations as defined in Rosette is similar to that of generic functions in CLOS [Gabriel et al. 1988]. A message is sent to an operation and the operation determines the next step in getting the work accomplished. Rosette provides a number of built-in operations called *primitives* that implement basic behaviors on the entities in the initial environment. For example, consider **`(+ 3 4)`**, the **`+`** is an operation that determines what action to take based on the arguments sent to it in the message. In this case, there are two **`Fixnums`** and the operation selects the **`Fixnum`** addition primitive.

The way the selection is done depends on the kind of operation. Typically, the operation will use the first argument as a target and send a message to the target that includes the operation as the key for the method to be executed. In general, this will not be the case. The operation **`+`** may be implemented so as to make a more direct determination of the method for addition. Further, in some cases the operation may make its determination based on the classes of more than one of its arguments, selecting a procedure specific to a particular combination of arguments. The syntax of putting the operation as the target in a communication is similar to traditional function-call syntax, and serves to integrate function invocation and method invocation into a common form of expression. Currently there are two kinds of operation supported in Rosette. The first is just called an **`Oprn`**, and **`+`** above is an example. The second kind of operation is called a **`SyncOprn`** after its behavior as a "synchronous" operation. A **`SyncOprn`** will cause a method to be looked up and invoked regardless of whether the target entity is locked or not. On the other hand messages sent as a result of an **`Oprn`** obey the usual Actor semantics. There are two uses for **`SyncOprn`**s. The first is to implement the initialization protocol associated with creating an instance. This is typified by the **`init`** operation. The second use of **`SyncOprn`**s is to implement diagnostic operations that permit examination of entities regardless of their communication state. An example of this use is the synchronous operation **`messages`** that retrieves a tuple of the messages awaiting any entity.

The following is a list of identifiers that are bound to operations in the initial environment:

```
=               >               <
!=              >=              <=
+               -               *
%               /
->              ->fixnum        ->float
->tuple         ->string        ->symbol
abs             cdiv            ceil
clear           concat          cons*
```

methods. Prototypes are entities that are copied to create new actors and objects that inherit from a shared behavior object. A prototype is a representative of a class of entities. The search for the method to be used when a message is received starts at the receiving entity and continues with its parent, which is typically an SBO. **SendExpr** is an example of a prototype, and its associated shared behavior object collects together all of the methods that apply to send expressions such as accessors of the components of a send expression and the method for expanding a send expression. The following is a list of the names of the prototypes in the initial environment. As a convention, the names for prototypes start with an upper-case letter.

#absent	Actor	BlockExpr
Bool	Char	Code
CodeVec	ComplexPattern	ConstPattern
Ctxt	EmptyExpr	EmptyMbox
#eof	Fixnum	Float
FreeExpr	GotoExpr	IdPattern
IdAmperRestPattern	IdVecPattern	IfExpr
IndexedMeta	Istream	LabelExpr
LetExpr	LetrecExpr	LockedMbox
Meta	Method	MethodExpr
MIObject	Monitor	#niv
	Oprn	Ostream
Prim	Proc	ProcExpr
Queue	QueueMbox	QuoteExpr
RequestExpr	SendExpr	SeqExpr
SetExpr	Stack	String
Symbol	SyncOprn	Table
Template	Timer	Tuple
TupleExpr		

Some of these prototypical objects were introduced in section **4.1** on constants. Several will be discussed briefly at this point, and the remainder in later sections of the report. The prototypical **Actor** is used to create new families of entities. The prototypical **MIObject** is used to create shared behavior objects that provide for multiple inheritance. The prototypes **Meta** and **IndexedMeta** are used implicitly by the Rosette system in the process of creation, and represent objects that describe the structure of other entities. Every entity has an associated meta object that may be accessed via the **meta** primitive. The prototypical **Prim** is included for completeness but it is not possible to create new primitives by simply cloning it. Primitives can only be created by extending the underlying Rosette implementation. The prototypes **Method**, **Proc**, **Code**, and **CodeVec** are included also for completeness. They may be cloned to produce new instances although they are usually created as a result of the **compile** primitive on **Exprs**. New methods can be added at any time to extend the behavior of any of the above kinds of entities. The slots representing their local state are freely accessible to Rosette code.

4.2. Variables and definitions

⟨*id*⟩ basic syntax

Any identifier that is not a keyword may be used (unambiguously) as a variable. A variable may name an actor or object which is said to be bound to the variable. The set of all such bindings in effect at some point in a program is known as the environment in effect at that point. The actor or object bound to a variable is called the variable's value.

Like Algol, Pascal, and Scheme, Rosette is a statically scoped language with block structure. For each bound variable, there is a region of the program text within which the binding is visible. The region is determined by the particular binding construct. Every reference to a variable refers to the binding of that variable established by the innermost region containing a binding of the variable. There is an outermost region, the "top-level environment", and if a variable is not bound in any region including the top-level environment, then it is said to be unbound.

The construct **`define`** introduces new variable bindings in the top-level environment.

(`define` ⟨*id*⟩ ⟨*expr*⟩**)** ⇒ ⟨*id*⟩ derived syntax

The ⟨*expr*⟩ is evaluated and the result bound to the ⟨*id*⟩. The defining form returns the symbol that was defined.

An expression consisting of a variable is a variable reference. The value of the variable reference is the value bound to the variable in the environment in effect at the point of the variable reference. An exception will be raised if a variable reference is made to an unbound variable.

```
(define x 28)                              ⇒ 'x
x                                          ⇒ 28
```

Other binding constructs are as follows: **`proc`**, **`method`** and **`pure`**, which bind formals to actuals (section **6.2**); **`let`**, **`let*`**, and **`letrec`**, which introduce local bindings typically for the purpose of temporaries in definitions (section **5.3**); **`do`**, **`do*`**, and **`iterate`** which introduce local bindings in looping constructs; and the form **`defActor`** , which generates shared behavior objects and their associated prototypical actors that handle multiple types of requests. Actors are entities that have a local state that may be modified in response to messages that they receive (see below and section **6.3**).

4.3. Prototypes, operations and primitives

4.3.1. Prototypes and shared behavior objects

A shared behavior object (SBO) is much like a method dictionary in Smalltalk-80 [Goldberg and Robson 1983]. An SBO collects together the methods and other values that are defined over copies (instances) of the associated prototypical actor or object. All of the instances share common definitions of these concatenation of the preceding actors and the tuple actor resulting from the last expression. The two following examples illustrate the difference in the two forms of writing tuples:

```
[3 2 [1 2 3]]                        ⇒ [3 2 [1 2 3]]
[3 2 & [1 2 3]]                      ⇒ [3 2 1 2 3]
```

The "`&` ⟨*expr*⟩" form flattens the tuple resulting from ⟨*expr*⟩ into the tuple being constructed. The operations available on tuples are described in section **7.6**.

Strings are written as follows:

"⟨*character*⟩ …" basic syntax

A string is a sequence of 0 or more characters. They are used for example in input/output operations and to construct identifiers in programs. The escape sequences defined in section **3.2** may be used to include certain non-graphic characters as well as the characters "\" and """ in strings. Operations on strings are described in section **7.5**.

```
"a \" in a string"
```

Expression literals are written using the "'" character:

'⟨*expr*⟩ basic syntax

An expression literal is an actor that represents an expression as data. There is a rich set of expression actors corresponding to the different sorts of expressions in Rosette (see sections **7.7** and **8.1**):

```
BlockExpr   FreeExpr   GotoExpr IfExpr   LabelExpr LetExpr
LetrecExpr  MethodExpr NilExpr  ProcExpr QuoteExpr RequestExpr
SendExpr    SeqExpr    SetExpr  Symbol   TupleExpr
```

(The identifier `Symbol` is used to name the prototypical identifier expression.) The basic input operations (section **11**) parse and return expression actors, as does macro expansion (section **10**).

```
'a-symbol                            ⇒ a-symbol
'[a 2 3]                             ⇒ [a 2 3]
'(+ a 3)                             ⇒ (+ a 3)
```

The second example above is a **`TupleExpr`** while the last example is of a **`RequestExpr`**. Note that a **`TupleExpr`** is not the same as a tuple. The evaluation of a **`TupleExpr`** produces a tuple, while the evaluation of a **`RequestExpr`** can produce any sort of value. Operations on expression objects are discussed in section **8.1**. Discussion of the use of expressions in the definition of actors and objects follows in the remainder of this section.

and	**become**	**break**
cond	**dec**	**define**
defActor	**defExpander**	
defOprn	**defMethod**	**defProc**
defPure	**defSlot**	**defSlots**
defSync	**defValue**	**do**
do*	**inc**	**iterate**
local	**m!**	**next**
not	**method**	**or**
proc	**pure**	**set**
slot	**super**	**update**
value	**void**	**s!**

4.1. Constants

Constants are actors that always have the same behavior. Several classes of constants are built-in. They include familiar atomic data types, the basic constants and the structured constants. Structured constants have components that may be accessed. These include tuples, strings, and expressions.

4.1.1. Basic constants

⟨*basic constant*⟩ basic syntax

The kinds of basic constants are **Boolean**, **Character**, **Fixnum**, **Float**, and the following special constants: **#absent**, which is used to indicate that no value is present, **#eof** which is used to signal that the end of a stream has been reached, and **#niv** (no-intrinsic-value), which is used when no other value is appropriate. Operations on basic constant actors are described in sections **7.1-4**. The basic constants evaluate to themselves:

```
#t          ⇒ #t
#\b         ⇒ #\b
231         ⇒ 231
3.14        ⇒ 3.14
#absent     ⇒ #absent
#eof        ⇒ #eof
#niv        ⇒ #niv
```

4.1.2. Structured Constants

Structured constants include tuples, strings, and expression literals.

Tuples are written as follows:

[⟨*expr*⟩ …**]** basic syntax
[⟨*expr*⟩ … **&** ⟨*expr*⟩**]** basic syntax

Tuples are used to represent messages that convey information between two actors in a communication, to introduce sub-structure in messages, and as a primitive data structure in the definition of other actors. The first form above permits 0 or more actors to be packaged up in a tuple. The second form requires that the expression following the **&** must evaluate to a tuple, and the resulting tuple is the

4. Simple constructs

Constructs may be characterized as either simple or compound. Simple constructs are used to build larger constructs called compound constructs. This section defines the simple constructs available in Rosette. Section **5** describes the compound constructs.

An expression is a construct that, when evaluated, returns one or more values. Examples of classes of expressions include constants, variable references, requests, and conditional expressions. A command is a construct that sends one or more messages to initiate actions at other actors. Expressions and commands may be used interactively with a system supporting Rosette; however, they are more frequently used to describe the behavior of actors in response to the receipt of messages.

Expression and command constructs may be categorized as either basic or derived. Basic constructs include variables, request expressions, literals, and special forms. Special forms are converted by the Rosette reader to instances of the appropriate expression type. Derived constructs are not semantically primitive, but can be explained in terms of the basic constructs (see Appendix 2). They are redundant, but capture common patterns of usage, and are provided therefore as convenient abbreviations. Moreover, they may be implemented in a direct fashion either in an interpreter or compiler as warranted by performance considerations.

Special-form constructs and the derived constructs are written using request expressions with a syntactic keyword as the target of the request. A Rosette program is first macro expanded and then compiled and run. The expansion is performed in an environment in which the syntactic keywords are "bound to" procedures that perform macro transformations on Rosette programs. The following special-form keywords:

```
block          free           goto
if             label          let
let*           letrec         method
proc           send           seq
set!
```

and the following derived (macro) keywords are defined in the initial Rosette environment:

#niv	the “no intrinsic value” constant
#absent	the constant indicating an absent value
#eof	end of a stream

; Semicolon starts a comment except when it occurs within a string. The comment continues to the next newline or page break in the input stream. Comments are treated as a part of the newline or page break and consequently may not appear in the middle of a token.

The escape character “****” is used to write some common non-graphic characters that may be used as characters by themselves or included in symbols or strings. In particular, the escape sequence “**\\ **” is used to include a space in a symbol. An example of its use is: “**symbol\\ with\\ spaces**”. It is legal, though not necessary, to use it when writing a character constant or string. “****” is used to write the backslash character itself. Thus the character constant backslash is written “**#**”. The following are additional escape sequences:

\n	newline
\r	return
\t	tab
\f	page break
\\"	string quote
\x*dd*	hexadecimal character code

Examples of the above are:

```
"a string\ with a \"" ≡ "a string with a \""
"a\tstring\tthat tabs and returns\r"
"the page break character is written \"#\\\\f\""
```

3. Lexical conventions

This section introduces the syntax of the tokens (out of which expressions are built) in Rosette programs.

3.1. Identifiers

Rosette identifiers follow conventions similar to those of other languages. An identifier may be formed from a sequence of letters, digits, and "extended alphabetic characters" that begins with a character that is not a digit. The following are extended alphabetic characters:

```
+ - * / < = > ! ? $ % _ ~ ^
```

Examples of identifiers are:

```
proc                     q
matrix->determinant      beef-Broth
+                        <=
```

Upper and lower case forms of a letter are always distinct. Thus `foo`, `Foo`, and `FOO` are all distinct identifiers. An identifier is terminated by white space or one of "`(`", "`)`", "`[`", or "`]`". White space characters are spaces, tabs, newlines, and page break. White space is used to improve readability and to separate tokens from each other as necessary, but is otherwise insignificant. (A token is an indivisible lexical unit such as an identifier or a number.) White space may occur between any two tokens, but not within a token. White space may also occur inside a string literal, where it is significant.

Identifiers in Rosette may be used as syntactic keywords (see section **4** and **10**) or as variables that are bound to a value (see section **4.2**). When an identifier appears as a literal or within a literal (see section **4.1**), it represents a symbol (see section **7.7**).

3.2. Other notations

The following summarizes the other notations used in Rosette:

`0...9`	Digits are used to notate numbers in base 10.
`( )`	Parentheses are used to notate sending communications (see section **4.4**).
`[ ]`	Brackets are used to notate tuples (see section **4.1**).
`'`	Single quote is used to indicate literal expressions (see section **4.1**).
`"`	Character string constants are delimited by double quote characters (section **4.1**).
`\`	The backslash introduces character escapes in symbols, characters, and strings (section **4.1**).
`#`	The sharp sign is a prefix for a variety of constants (section **4.1**): `#t #f` boolean constants `#\` character constants

that may be stored as the value in a slot just as any other actor. Thus, the concepts of acquaintances and script are unified in Rosette. Further, by breaking the script up into methods, inheritance becomes a reasonable way to support sharing and code reuse within the context of the actor model. Two other hidden or meta slots associated with every actor in Rosette are **`meta`** and **`mbox`**. The **`meta`** slot of an actor refers to another actor that describes the structure of the first actor, while the **`mbox`** slot refers to an actor that implements the mail-queue abstraction (see section **9**).

2.2. Objects

In this version (1.1), the distinction between objects and actors has been removed.

2.3. Environments

There are two environments that are important to know about in the Rosette system: **`Global`** and **`Top`**. Essentially, **`Top`** is the top-most object in the inheritance graph in which shared slots are stored. It is not necessary for the inheritance graph to be rooted; however, the majority of the entities that are defined in the system are in fact rooted at **`Top`**. The **`Global`** object is an environment that maintains bindings for symbols that are intended to be accessed from the Rosette listener loop and serves as the final environment that the compiler will check when resolving a free-variable. Typically, symbols that are bound to operations will occur free in a method for an entity and are resolved against the **`Global`** environment.

directly in messages. Since primitive actors are immutable, their identity may be represented by their state. That is, `3` is a sufficient identity for itself since its behavior is the same always and everywhere.

A non-primitive actor has an identity that is represented by a *reference* and a current *behavior* that is composed of a set of acquaintances (the instance variables or local state) that include the methods that define the actions that the actor can take upon receipt of a message. The acquaintances of an actor are other actors that messages may be sent to or may in turn be sent in messages. When a non-primitive actor is sent in a message, it is actually the reference that is sent. Figure 1 illustrates an idealized view of a non-primitive actor. The basic form of interaction between actors is asynchronous-buffered communication. Thus, associated with every actor is a mail-queue that serves to buffer communications that are sent to the actor until they can be accepted for processing. In addition, one or more behaviors may be associated with the actor, thus a form of pipelining is intrinsic to the actor model. The pipelining arises from the characteristic of the semantics that it is possible for the replacement behavior to be established while other actions are being taken with respect to a received message.

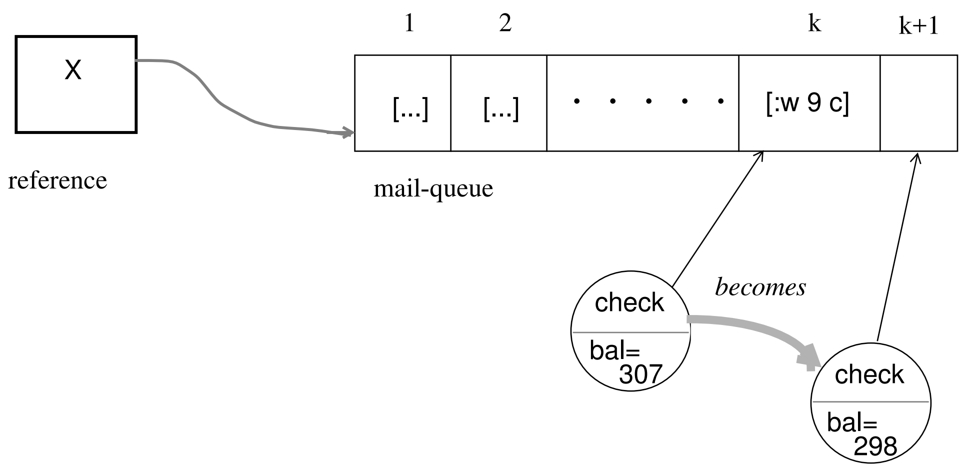

Figure 1

When an actor receives a message, it is locked until it explicitly establishes its state and behavior for the next message to be received (i.e., its replacement behavior). Further, in response to a message, an actor may send messages to other actors it knows about and also create new actors.

In Rosette, the structure of an actor is concretized as follows. Every actor is considered to be a collection of slots that may be viewed as key-value pairs. Any actor whatsoever may be used as a key. Further, every actor has a hidden or meta slot that refers to its **`parent`**. If an attempt is made to lookup a key in an actor and no slot of the actor has that key, then the lookup continues with the parent actor. The concept of an actor's script is made concrete by introducing method and procedure actors

1. Introduction

This report describes the Rosette language relative to release 1.0 of the Rosette system. The Rosette language is a concurrent object-oriented programming language. It is prototype-based and incorporates multiple inheritance and reflection. The Rosette system provides an environment in which concurrent object-oriented programs may be written and run on an ideal multi-computer system consisting of unlimited numbers of processors and unit communication delays. The model of concurrency implemented in Rosette is that of the Actor model [Agha 1986].

This document is organized as follows. Section **2** provides a brief overview of basic concepts. Section **3** presents the lexical conventions used in writing Rosette programs. Section **4** presents the simple expressions and commands. Section **5** defines the compound constructs. Section **6** discusses how to define new actors. Section **7** presents the operations available on primitive actors. Section **8** describes operations on program representation actors, section **9** presents the reflective model and monitoring facilities, section **10** discusses the mechanisms for syntactic extensions, and section **11** presents basic input/output facilities.

2. Basic Concepts

2.1. What is an Actor?

In this model, everything in a system is taken to be an actor. In this respect the model is uniform in the same way that Smalltalk takes everything to be an object. Two important differences are that in the Actor model, each actor is active in a manner that is completely independent of all other actors and further, all of the actions taken by an actor upon receipt of a message are concurrent. That is, there is no implicit serial ordering of the actions in a method (or script as it is referred to in the Actor model). This approach readily lends itself to descriptions of maximal concurrency.

All actors are characterized by an identity and a current behavior. Once created an actor's identity does not change even though the way that it behaves over time may. The current behavior of an actor represents how the actor will respond to the next message that it receives.

Actors may be partitioned into primitive and non-primitive classes. Primitive actors are used in the model to avoid a conceptually infinite regress of message passing. They correspond to the usual atomic types such as numbers and characters. An implementation will give direct treatment to passing messages to a primitive actor. For example, an implementation will directly interpret passing the message `[+ 4]` to the actor `3` by identifying the operation and performing it. Primitive actors are sent