EASEL

Language Reference Manual and Author Guide

EASEL

Survivability Simulation

Learn more at www.pbtomega.com

Software Engineering Institute ▪ Fifth Avenue ▪ Pittsburgh, PA 15213

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue," excludes openly gay, lesbian, and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone 412-268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone 412-268-2056.

Carnegie Mellon University publishes an annual campus security report describing the university's security, alcohol and drug, and sexual assault policies and containing statistics about the number and type of crimes committed on the campus during the preceeding three years. You can obtain a copy by contacting the Carnegie Mellon Police Department at 412-2682-2323. The security report is available through the World Wide Web at http://www.cmu.edu/security/stats.html

Obtain general information about Carnegie Mellon University by calling 412-268-2000.

CarnegieMellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Easel Language Reference Manual and Author's Guide

CMU/SEI-2002-TR-013

Alan Christie
Dan Durkee
David A. Fisher
David A. Mundie

2003-02-24

Version 3.0b3

Network System Survivability Program

Acknowledgements

This work was sponsored by the U.S. Air Force Electronic Systems Command and by the U.S. Army Telemedicine and Advanced Technology Research Center.

Contributors

The Easel design and implementation effort was carried out primarily by the Easel team within the Network System Survivability Program, namely David Fisher, David Mundie, Alan Christie, and Dan Durkee. Howard Lipson also participated in the design effort. Tim Shimeall played a very special role both during the design and as an early adopter. Numerous students from Carnegie Mellon participated in the implementation, notably Marcelo Cataldo, Steve Protulipac, Mark Firth, David Dixon, and Nawaporn Wisitpongphan.

No Warranty Statement

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Preface

Why Easel?

The past few years have shown a need for fundamentally new approaches to security and survivability of large scale networked systems. Infrastructures and other modern systems pose problems of dealing with partial information, complexity of combinatorial interactions of very large numbers of human and automated participants, ubiquitous access, loss of centralized administrative control, the growing threat of automated attacks and a shift in priorities for security from confidentiality of information to availability of services. -Survivability research seeks new methods including emergent algorithms, diversity and dynamic trust validation to ensure that systems satisfy their most critical requirements. Easel is an automated simulation tool for research in survivability, infrastructure assurance and other applications that must contend with large numbers of interacting components, lack of central control, and incomplete and imprecise information.

Easel is a programming language for simulating systems for which only incomplete or imprecise information is available. It allows abstract descriptions of anything. The things described may be real or imaginary. They may be physical, abstract, or computational. Easel is intended as a research tool for describing and simulation unbounded systems, emergent algorithms and survivability architectures.

Easel is a property-based language. It computes on accurate but incomplete abstractions in the form of property-based types.

Who should use this Manual?

This manual has two main audiences. First, the manual is being used as a specification document to support the Easel development team and others who are interested in semantics of the language. Secondly, it is aimed at those who wish to write simulations or other programs in Easel. In the body of the manual there are therefore two threads. The first thread, on white background, gives the signatures and semantics of the Easel operators. The second thread, on a gray background, gives a fuller commentary on the rational and usage of the operators, and provides illustrative examples of their application.

Conventions Used in This Manual

Easel is a work in progress - and this manual reflects the evolving nature of the language and its implementation.

Examples in this manual are of two types: complete executable programs, and program fragments. Some of the examples are not working in the current version, but are planned for future versions; they are marked as "planned". The fragments are marked as "fragment"; they have been checked for parse errors, but have not been tested by being embedded in a running program.

Note that there are several reasons an example might not work even though the operator it illustrates is implemented: the example might use unimplemented features of this operator, or might use other operators that are not implemented, or there might simply be a bug that prevented it from running at the time of writing.

Operators whose name appears in quotes, and operators whose name begins with an underscore, are special. Usually they are intended for internal use, and should not be referenced by the end user. For example:

```
"_overload_resolve"(any): any;
```

This is the definition of the overload resolution phase of the Easel translator, and it should not be called from user programs.

The examples in this manual uniformly use indenting to indicate program structure; see A.8 for writing programs using braces.

The signatures for many operators contain comments which indicate how the operator might be implemented; for example

```
centimeters: unit Length #{is meters / 100}#;
```

Here "is meters/100" is a possible body for the centimeters unit. It is enclosed in the Easel multiline comment delimiters "#{" and "}#" because Easel's support for dimensional analysis is not working in the Beta release.

Colophon

Most of the Easel project documentation is stored in the form of an XML project database. The current document was extracted from that database using the Apache tools *Xalan* and *FOP* along with a custom XSLT stylesheet. The fact that the *FOP* processor is still under development has led to some quirkiness in this PDF version, such as the Table of Contents appearing at the end and the presence of widows and orphans.

Chapter 1. A Bird's-Eye View of Easel

Prediction is very difficult, especially about the future. - Niels Bohr

In this section we will review some of the ideas that motivated the development of Easel. Central to this motivation is the ability to anaylse and engineer complex systems that are inherently survivable. To this end, we will look at the notions of emergent algorithms and unboundedness and how they have influenced Easel's design. We will look at the concept of types and finish the section off with a simple program that illustrates some of Easel's main features.

1.1. Unbounded Systems and Emergent Algorithms

Easel is a language designed to model systems where unboundedness and emergence are central themes. An unbounded system is any system in which the participants (human or computerized) have only incomplete or imprecise information about the system as a whole. They include human participants as well as automated components. Their boundaries are not precisely known. Interconnections among participants in unbounded systems change constantly. Furthermore, the trustworthiness and often the identity of participants is unknown. Centralized administrative control cannot be fully effective in such systems. These are the characteristics critical national infrastructures, of the Internet and of electronic commerce. They characterize most social, economic and biological systems, and most activities one participates in every day . Such systems contrast dramatically with the assumptions of closed centrally-controlled computer systems and with the assumptions underlying many modern computer security technologies.

Emergent algorithms differ from conventional hierarchical and distributed algorithms; they operate in the absence of complete and precise information, do not have central control, hierarchical structure or other single point vulnerabilities, and achieve cooperation without coordination. Mission requirements are satisfied in the form of global system properties that emerge from the combined actions and interactions of all system components. For reasons of mission survivability, our research considers only emergent algorithms that do not have single (nor any fixed number of) points of failure. For reasons of practicality and affordability, we consider only those emergent algorithms in which the cost of each node (whether measured in dollars, cpu cycles, storage requirements or communications bandwidth), is less than proportional to the number of nodes in the system. The effectiveness of this approach can be further enhanced by dynamic trust validation among the participants.

1.2. Security and survivability

A major motivating factor behind the development of Easel was the need to address survivability issues. Survivability is a relatively new research domain that seeks new techniques for mission fulfillment in unbounded systems such as the Internet, electric power system, telecommunications, military command and control, and health care infrastructure. The role of infrastructure assurance is to ensure survivability and fulfillment of these missions. It is about managing business risks to satisfy critical mission requirements in the face of competition and adversity. It is about making trade-offs to ensure continuity of services in the absence of complete and precise information. Survivability is defined as the ability of a system to fulfill its mission in the presence of attacks, failures and accidents. Survivability and infrastructure assurance are not about adherence to industry standard security practices. They are about confidentiality, information integrity, correctness and dependability only to the extent that these particular quality attributes are the most critical requirements of the mission

Successful infrastructure assurance depends of well reasoned business decisions and cannot be achieved through technological solutions alone. Security technologies are often effective where the primary objective is confidentiality or integrity of information, where the system has a clearly defined perimeter with centralized administrative control, where administrators have complete insight into all aspects of the system, and where all components and participants within the system are known, authenticated and trustworthy. Perhaps most importantly, is the assumption that applications can tolerate a binary fortress model of security in which either all attempted intrusions are successfully repelled or the system as a whole is compromised. Security solutions are inadequate if even a few of these conditions are absent. Unfortunately, modern large scale networked systems, including critical national infrastructures, seldom satisfy any of these conditions.

Technologies based on fortress models of security are successful in transparent, trusted, centrally-controlled systems with clearly defined perimeters because they exploit those constraints of their environment. Success in survivable systems and infrastructure assurance likewise depends on exploiting the properties of unbounded systems to full advantage. It is unlikely that any large networked system can be molded to satisfy the constraints of fortress based security. In contrast with traditional security methods that attempt to build impenetrable walls around trusted insiders, a fundamental assumption of survivability is that no participant or component of a system is immune to all errors, failures and compromises.

Survivability research seeks solutions that exploit the inherent characteristics of unbounded systems to ensure survivable missions in the presence of compromised components. Unbounded systems offer opportunities for cooperation without the cost and vulnerabilities of coordination. They enable distributed specification and local optimization with their attendant resistance to standardized attacks. They facilitate robust and resilient solutions in which no component is essential and compromises of individual components will not cascade.

1.3. The Need for Emergent Simulation

Although the benefits of ad hoc development of emergent algorithms has been demonstrated, a rigorous process for deriving emergent algorithms from mission requirements is prerequisite to their widespread use in automated systems. Intuitions about the global effects of the local actions and interactions among large numbers of nodes are seldom correct. The problem of designing emergent algorithms is especially difficult, because it begins with the desired global properties and attempts to determine what simple combinations of local actions and interaction would produce those effects over time in a large scale network. An effective design methodology will depend on greater understanding of the influences of local action and interaction on emergent global properties and on the sensitivities of emergent properties to local variations. The obvious and probably only means to answer these questions is by simulation of emergent algorithms and of the unbounded systems in which they operate. This recognition has opened a new area of research for simulation of unbounded systems.

Current simulation systems do not produce accurate predictions of the behavior of unbounded systems. By definition, unbounded systems are incompletely and imprecisely defined. Thus, a simulation of an unbounded system must be able to produce accurate results based only on incomplete information. Current models, however, require complete information and thus are always built with assumptions or inaccurate information. The ability to operate on abstract specifications and simulate at various levels of abstraction is a long-standing need of many applications, but is not provided as a feature of existing simulation systems. Equally important, all object based models (both physical and computerized) are inherently inaccurate because they are based on complete representations as objects. This might be acceptable when dealing with small numbers of nodes or when great care is taken to differentiate between which modeling results are likely to be valid. Such remedies seldom if ever succeed in differentiating inaccurate results when modeling complex or large scale system. Furthermore, as the number of subsystems in a model increases, the inaccuracies of each subsystem pervade the whole after a few iterations and guarantee that all simulation results will be inaccurate. This may account for the pervasive failure of large scale simulations to produce accurate results. These problems are aggravated in unbounded systems where the numbers of components are very large and a primary purpose of simulation is to accurately predict the global effects from local activities. Because accuracy and completeness are not simultaneously achievable when describing the physical world, accurate simulation is feasible only if the simulator can guarantee accurate results from accurate but incomplete specifications. Other difficulties in simulating unbounded systems include:

a. the need for thousands to tens of thousands of nodes per simulation,
b. lack of linguistic mechanisms in programming languages for making incomplete and imprecise specification,
c. the inability of object oriented computations to describe and reason about the real world,
d. the need to combine information about a system from multiple knowledge domains,
e. management of multiple simultaneous beliefs of the various stakeholders in an infrastructure,
f. integration among separately developed simulations, and
g. exponential increases in computational cost that accompany linear increments in the granularity or number of nodes in a simulation.

1.4. The Easel Solution

These considerations have led to a new approach to simulation called Easel, an emergent algorithm simulation language and environment. Easel employs a paradigm of property-based types (i.e. describing abstract classes of examples by their shared properties) to simultaneously address all of the above simulation problems. Because Easel is property-based it can be used to give accurate, but incomplete, descriptions of anything. In combination with an appropriate automated logic system, it can be used to produce accurate conclusions about examples from the physical world. This contrasts with physical models and automated simulations that depend on representation of objects, where descriptions must be complete (and thus inaccurate) and in which conclusions are accurate only for the model but never for their extensional interpretation in the real world.

While traditional modeling and simulations systems answer all questions without a mechanism for the user to determine which answers are accurate, Easel will report what additional information is needed to continue toward an accurate result. Easel will also supports multiple levels of abstraction, multiple simultaneous belief systems, distributed specification and dynamic graphic depictions. Easel is a discrete event simulation language plus limited support for continuous variables. The linguistic limitations of traditional programming systems for incomplete and imprecise description are overcome by use of quantifiers, adjectives, improper nouns, pronouns and other forms of anonymous reference in the language. In combination with property-based types, these mechanisms provide a semantic framework of examples of any type whether real or imagined and whether from the computational, mathematical or physical worlds.

In summary, we believe that concepts of local action and visibility, unboundedness, and emergent properties are important factors in simulating systems where large numbers of loosely coupled actors are involved. To this end, we have developed a new general-purpose simulation language called Easel, which is currently being implemented. The rest of this section summarizes the concepts behind Easel, describes some of its unique properties, and provides a simple example of Easel's use - from the ant kingdom.

1.5. A Type Is a Type Is a Type...

Being a property-based type (PBT) language, Easel considers all entities to be types. A **type** is a description of some class of objects, while a description is a set of properties. An example of a **type** is any object that satisfies the type's properties. Easel has a built-in family of types that can be extended to specify user-defined types. Figure 1 illustrates some of the important built-in Easel types and their relationships.

Types are built up by inheriting properties of parent types. The following are some characteristics of Easel types:

a. All types inherit from the root type **all.** This includes all consistent (true) **types** and inconsistent (false) **types**.
b. The **type** that contains all **types** is the false type. It is so named since types with inconsistent properties (e.g., 3 > 5) are also **types**. However, the false **type** has no examples.
c. As one ascends the **type** hierarchy, **types** accumulate more and more examples from their children.
d. As one descends the **type** hierarchy, **types** accumulate more and more properties inherited from their parents.

Here are some representative Easel types:

a. Mutable: a type whose properties are changeable
b. Immutable: a type whose properties are unchangeable
c. Actor: a physical "thing" that has behavior and is threaded
d. Abstract: a type that can be described completely within computer memory
e. Type: the set of named types is also a type (the type **type**)

Even the number 5 is a type, albeit a degenerate, singleton type: it inherits properties from the types positive integer and odd integer, among others.

1.6. Type Manipulation Within Easel

The Easel type hierarchy allows you to build up and manipulate types. This is unlike other conventional programming (or simulation) languages where you can only operate on examples of the type (e.g., through iteration). In the object-oriented paradigm, the equivalent to types is classes. However, Easel's types can be manipulated in powerful symbolic ways that are not possible with classes. The program shown in the example below illustrates some of the ways that types can be manipulated (note that this is a program, not a simulation).

```
# STATUS: planned

dog_gonned( ): action is
        food: type is enum(meat, vegetable, anything);
        dog: type;
        omnivorous: type is
                diet :: food := anything;
        Corgi: type is
                property dog & omnivorous;

        Jenkins: Corgi;
        confirm Jenkins isa omnivorous dog;

dog_gonned();
```

As is usual in Easel, we first define some simple types, and then build up more complex types that inherit the properties of these simple types. Thus we begin by defining the type `food`, which has `vegetable`, `meat`, and `anything` as values, and the type `dog`, which has no specified properties. Then we build up the adjectival type `omniverous`, which is the type of all creatures with a diet of `anything`, and the type `Corgi`, which is an omnivorous dog. Finally we declare Jenkins to be an **attribute** whose type is `Corgi`, and confirm that he is an omnivorous dog. Easel uses **:** to define constants (see 3.2.1), and **::** to define attributes whose value may vary over time (see 6.2.1).

Parameterized types allow subtypes to be defined in the same way that adjectives qualify nouns in English to select subtypes.

```
# STATUS: planned

example( ): action is
        flower: type;
        red(any): type;
        large(any): type;
        rose: type is large red flower;
        American_Beauty: rose;

        confirm American_Beauty isa red flower;
example();
```

Here `large red flower` is an adjectival phrase that returns the subtype of `flowers` that are `large` and `red`. This use of adjectives provides Easel with a powerful mechanism for defining subtypes.

In a similar way, the use of quantifiers such as **any**, **all**, **some**, and numeric quantifiers provides Easel with a means of selecting subtypes that have to be manipulated or tested.

```
# STATUS: planned

example( ): action is
        flower: type;
        red(any): type;
        large(any): type;
        bouquet: list flower := 3 new large flower;
        biggy: type is all large flower;
example();
```

Here `3 new large flower` is a quantified expression that produces a list of three

```
large flowers.
```

1.7. Actors and Neighbors

Easel's architecture is designed so that it can simulate very large numbers of independent **actors**. **Actors** are simulated entities of the physical world (e.g., a system administrator, user, intruder, automobile, bird, or the moon), of the electronic world (e.g., a computer, router, or peripheral device), or of the software world (e.g., a software agent or task). Giving each **actor** its own thread of control allows for a high degree of parallelism in Easel's execution. **Actors** can interact directly only with their near neighbors, and only in ways prescribed by their neighbor relationships. Neighbor relationships are protocols of interaction and are defined as types that can be associated with **any actor**. Thus, in a simulation of birds in flight, a bird's near neighbors might be **any** bird or other object that the bird can see from its current position and heading. In an organizational simulation, an **actor's** near neighbors might be only those actors who are connected by formal organizational ties, and neighbor operations might include sending and receiving messages.

In summary, actors have threads of control, have behavior, are "born" and can "die" , and have significant performance advantages over non-threaded approaches.

Many existing discrete and continuous simulation languages use statically specified nodes and links in constructing models - the associations between the elements of their models are predetermined. This is appropriate for processes that have fixed topologies, such as those in the manufacturing paradigm. However, in many situations, dynamically changing relationships between **actors** (as reflected, for example, by their changing distance from each other) require a more general approach. In this regard, Easel introduces a neighbor relationship concept. A **neighbor** can be a function not only of physical distance, but of other considerations such as network linkage, organizational dependencies, or other properties in common.

1.8. Interacting with a Simulation

A simulation needs mechanisms through which it can be controlled. Thus we need to be able to start the simulation, set up simulation parameters, change parameters while the simulation is proceeding, and observe output from the simulation. None of these functions are part of the simulation: they either send information to the simulation or retrieve it from the simulation. Easel recognizes two distinct roles: facilitators, which allow for global control, introduction of new examples, and control of parameter values, and observers, which extract values of the simulation parameters to do statistical analysis or to drive graphical depictions.

Easel's visibility rules allow selected actors to play the role of facilitators and observers without any extra mechanisms.

1.9. An Example of a Simple Easel Simulation

As a gentle introduction to Easel, let's consider the following example. Ants are an interesting example of emergent algorithms, because an ant colony as a whole can evince behavior that is complex, but without global visibility and central control. As a contrived, entomologically unrealistic example, consider the question of how ants could form a circle without any communication among them. Here as an Easel simulation that shows how it might be done:

```
    # STATUS: current
```

```
# We are simulating an ant hill
ant_hill: simulation type is                                   # 1
        v :: view := ?;
        ant_list :: list := new list any;

# Life cycle of an ant
ant(id: int): actor type is                                    # 2

        # Ants have a position and a heading
        heading :: number := random(uniform, 0.0, 2.0*pi);     # 3
        x :: number := 250.0;
        y :: number := 250.0;

        # Create a depiction
        depict(sim.v,                                          # 4
                var offset_by(paint(circle(0.0, 0.0, 5.0),
                (firebrick)), var x, var y));

        # Pick a heading, walk out, walk back, repeat
        for every true do                                      # 5
                heading := random(uniform, 0.0, 2.0*pi);
                for h: each (1 .. 20) do
                        x := x + 10.0 * cos heading;
                        y := y + 10.0 * sin heading;
                        wait 1.0;
                heading := heading + pi;                       # 6
                for h: each (1 .. 20) do
                        x := x + 10.0 * cos heading;
                        y := y + 10.0 * sin heading;
                        wait 1.0;

# Create the simulation and start its actors
ant_circle(n: int): action is                                  # 7

        # Create an ant hill and its view
        ant_sim :: ant_hill := new ant_hill;                   # 8
        ant_sim.v := new view(ant_sim,
                "Ant Circle", (papayawhip), nil);
        null make_window(ant_sim.v, 0);

        # Create the ants
        for i: each (1..n) do                                  # 9
                push(ant_sim.ant_list, new (ant_sim, ant i));

        # Wait for simulation to finish
        wait ant_sim;                                          # 10

ant_circle 50;
```

We first declare a simulation type, `ant_hill` (at the line labeled 1). This simulation type has two attributes in addition to the predefined attributes of simulations: the `view v`, which is used to portray the ants, and a list of ants `ant_list` which can be used to reference ants (for example by iterating over the list).

At the line labeled 2, we declare the `ant` **actor** type. The **actor** is a predefined type in Easel that has the property of being threaded. This means that an **actor** is an independent process, has its own memory allocated (while it exists), and requires some CPU time to update its internal state and interact with other actors.

The properties of the `ant` type are defined next (point 3). These simple ants have only three properties: their orientation, and x and y coordinates. Note that the angle for any specific ant is defined randomly. Thus, each ant starts off at the center of the coordinate system with random initial orientation.

At line 4 we specify that each ant is to be depicted using a `firebrick` circle with a radius of 5 units, offset by whatever the `ant's` current coordinates are. It is the call on **var** that ensures that the depiction of the `ant` is continuously updated in the display as the ant's `x` and `y` coordinates change.

Starting at point 5 we provide the basic simulation loop through which the behavior of each ant is defined. This loop manages the thread of control for the `ant` in question and defines the `ant's` behavior. In this simple case, the `ant` moves out 20 steps, then turns around and moves back to the center

The `ant_circle` procedure starting at point 7 is the facilitator which manages the simulation. It first creates an `ant_hill` simulation and its `view` (at point 8), then allocates the `ants` (at point 9). Once a `new ant` has been created, it initiates a thread of execution that allow the `ant` to behave as specified by the `ant's` type (at points 3-6).

Some miscellaneous observations of the program may be worth making. Note the **type hierarchies**: `ant_hill` is a subtype of simulation type, while an `ant` is a subtype of the actor type. The code structure is defined through indenting and "outdenting." Comments extending to the end of the line are prefaced by the pound sign (#).

The ant example is interesting for a number of reasons. First, it demonstrates the concept of emergent properties. In this model, the emergent property is the circle that the ants generate as they move away from the nest. No individual ant has knowledge of the fact that it is part of this circle, yet, viewed globally, this is the shape that they collectively generate. Second, the example demonstrates the concept of **neighbors**. If neighborliness can be equated to nearness then, as each ant moves further from the nest, the number of its neighbors reduces, to the point that the ant finally has none. Third, other simulation languages that can address problems such as the ant circle do so using a grid pattern that constrains each ant to be located in one of the grid's squares. This granularity issue has ramifications, for example, in accuracy of representation. While Easel could model the problem using a grid, no such grid need be imposed as each ant simply has position as one of its properties.

Chapter 2. Types

2.1. Property-Based Types

Property-Based Types (PBT) is a system for describing categories of things by their properties. This contrasts with object oriented systems which describes classes by their examples (or more precisely, by representations of their examples). In a PBT system, a type is a set of properties. Each property is a predicate that is true for every example of the type. Anything that satisfies the properties of a type is an example of that type. A type may describe examples that are physical, such as people, for or the earth; that are abstract such as integers, greed or beliefs; or that are computational such as machine integers, data structures or the content of a disk.

More formally, a property based type is the transitive closure of a set of properties under implication in an intensional logic. Types provide a formal mechanism for accurately representing and reasoning about abstractions, informal ideas and incomplete and imprecise specifications.

Types are the primary objects of computation in Easel. Types are neither sets of objects nor data representations. Each type is incompletely described and thus may have unmentioned properties. The names of types are used like improper nouns in natural languages. Because any set of properties defines a type, there are an infinitude of types. Most types do not have names and instead are described by their differences (e.g., adding adjectives) from named types using a variety of operations including abstraction and specialization.

If s is a subtype of type `t`, then every property of `t` is also a property of `s` and every example of `s` is also an example of `t`. That is, a subtype has a subset of the examples and a superset of the properties of the **any** type of which it is a subtype.

2.2. Defining Types

Commentary

Types are defined using the **define** mechanism of 3.2.1.

```
# STATUS: current

t1: type;
t2: type is ?;
t3(int): type;
```

Here `t1` is defined to be a **type**. Because no body is given, the value of `t1` is unspecified - it is, in fact, equivalent to `t2`, which provides an explicitly unspecified body. The type `t3` illustrates that types may be parameterized.

2.2.1. type: immutable type;

Type is the type of all types including itself. Every type is an example of type **type**. Every type is also a subtype of type **type**. Each type is an immutable abstraction.

2.2.2. control "_scope"(property...): type;

The **scope** operation encloses the set of properties defining a type in a type definition. Scope may contain any number of properties. Each parameter to scope must be a property.

Attribute definitions (see 6.2.1) must appear before the statements (see 7). Attribute initialization and statements will be evaluated in the order of their appearance. Property assertions are noted and taken as true throughout the most local enclosing simulation throughout the life of the simulation.

Syntactically, a scope must be enclosed by **{ }** or by line indenting (see A.8).

2.2.3. control "_init_frame"(any): action;

This is an internal operation to reserve a space for for-loop control variables and attribute definitions inside compound statements. This operation is inserted by the semantic analyzer, which calculates the space needed for the control variables and attribute definitions.

2.2.4. control "_end_init_frame"(any): action;

This is an obsolete internal operation for actors. It marks the end of the attribute initialization phase (the elaboration, in Ada terminology). This operation is currently a no-op.

2.3. Types, Predicates, and Properties

As in natural language, **types**, **properties**, and **predicates** are closely related in Easel. Any named type can be used as a predicate testing for the satisfaction of the properties contained in that type. Any predicate can be used as a type operator applied to a type that returns the subtype of that type which satisfies the predicate.

Commentary

Consider the following two examples.

```
# STATUS: planned


car: type;
Japanese: type;
Honda: type is
        property Japanese car;
my_jalopy :: car := new Honda;
confirm Japanese car my_jalopy;
```

and

```
# STATUS: planned

car: predicate;
Japanese: predicate;
type Honda is
        property Japanese car all;
my_jalopy :: car := new Honda;
confirm Japanese car my_jalopy;
```

These two examples are exactly equivalent in Easel. In the first, `car` and `Japanese` are sets of unspecified properties, but they can be used to form new types and as predicates testing for satisfaction of those properties. (Even though we do not know anything about the properties implied by being a car and being Japanese, we know that my jalopy is Japanese because it is a Honda, and a Honda is defined as satisfying the properties `car` and `Japanese`.)

In the second example, `car` and `Japanese` are unary boolean-valued **predicates**, but they can still be used to form new types. When we write `confirm Japanese car my_jalopy`, we are calling the predicates `Japanese` and `car` on my jalopy, but the semantics is the same as if they were types.

2.3.1. property(cee type): property;

A supertype **property** asserts that all properties of the given type are also properties of the current type. That is, the given type is a supertype of the current type. Often the supertype will be an anonymous type specified as a named type specialized (i.e. using **&**) by some predicate.

(For the meaning of **cee**, see 7.9.1.)

Commentary

```
# STATUS: planned

even_posInteger: type is
        property int & it > 0 & it mod 2 = 0;
```

Note that here `it` refers to an arbitrary example of `even_posInteger`, and `&` computes the intersection of properties

Restrictions

The current implementation of **inheritance** is limited to attributes, including formal parameter attributes. In addition, computed attributes (e.g. `dog & it.brown`) are not allowed. However, the **type** operations **|**, **&**, and **~** are supported as long as their parameters are simple types.

2.4. Other Property Specifications

There are several other forms of property specification including attributed properties of each example of a type (see 6.2.1), relational properties among types and their examples (see 2.7.6), operational properties that describe the activities of types with dynamic examples (see 7), asserted properties that specify state relationships that prevail during these activities (see 7.6.1), autonomous properties of actors (see 8), neighbor relationships and interactions among actors (see 8.2), explicitly inherited properties from other scopes (see 2.3.1) and libraries (see 2.4.1).

2.4.1. include(string): any type;

The **include** operation reads the content of a file into the program at compile time. The file is specified by the `string` name of the file in the underlying operating system. The designated file must be a string that is a syntactically well formed structure of an Easel program.

Include would typically be used to include definitions.

Commentary

```
# STATUS: current

include "jumper.easel";

jumper_simulation(): action is
```

```
            s :: simulation := new simulation;
            for i: each (1..5) do
                    null new(s, jumper(i));

    jumper_simulation();
```

This example includes the file "jumper.easel", which might contain the following definition of a jumper:

```
    # STATUS: current

    jumper(id: int): actor type is
            x :: int := ?;
            y :: int := ?;
            for each (1..30) do
                    x := random(uniform, 0, 300);
                    y := random(uniform, 0, 300);
                    output("Jumper ", id, " at ", x, ", ", y, "\cr");
                    wait 1.0;
```

The effect of the include is as though the text in "jumper.easel" had been substituted in place of the include statement.

2.5. Lambda Types

Any named type may be parameterized. Several parameterized types may have the same name provided they have different numbers of parameters, have different formal parameter types, or produce the same results when applied to the same actual parameters. In some contexts they can also be distinguished by their return types.

Commentary

Any named type may be parameterized. Parameterized types follow the syntax shown above.

The following is an example of a "full" parameterized type specification:

```
    # STATUS: current

    f(i: int): int;
```

Often the formal parameter name can be omitted, and the formal parameter referred to using quantification:

```
    # STATUS: current

    f(int): int;
```

And of course some parameterized types have an empty parameter list:

```
    # STATUS: current

    f(): int;
```

Several parameterized types may have the same name provided they have different numbers of parameters or different parameter types, or produce the same results when applied to the same actual parameters. To illustrate, consider the following four parameterized type definitions:

```
# STATUS: current

f(i: int):int is
        return i + i;
f(k: int): int is
        return 2 * k;
f(s: string):int
        return length s;
f(int1:int, int2:int): int is
        return int1 + int2;
</main-example>
<variant-example kind="fragment" status="planned"><![CDATA[
f(int):int is the int + 1;
f(int & it < 0): int is -1 * *the int + the int;
f(string):int;
f(int1:int, int2:int): int is int1 + int2;
```

Although all four types have the same name `(f)` they may appear in the same program because they can be distinguished based on their parameter types. The types shown in the first and second lines have only one parameter, but they both return the same values given the same arguments. The type shown in the third line also has a single parameter, but its type is different. The type shown in the fourth line has a different number of parameters.

Parameterized types were first introduced in Chapter 1. However, the example below illustrates the use of parameterized types more fully. `Hair` is defined as having `color` and `length`, `mammals` as `warm-blooded hairy` things, and `dogs` as `mammals`. Then, specialized types of dog are defined. We can then test the truth (or falsity) of relationships between the defined types. Thus, we see that a `poodle` is a `brown dog`, and `brown dogs` are a subtype of the type `mammal`.

```
# STATUS: planned

example: type is

        # Build up inherited properties

        # Hair is colored and of a certain length
        hairs: type is
                shade: white | brown | gray | black := ?;
                hair_length: Length := ?;

        # Things that have hair are hairy
        hairy: predicate is
                return has(x, hair);

        # Things that have brown hair are brown-haired
        brown_haired: predicate is
                return has(x, hair) & x.hair.shade = brown;

        # Things that have hair longer than 15 cm are long-haired
        long_haired: predicate is
                return has(x, hair) & x.hair.hair_length > 15 cm;

        # Warm-blooded is a property
        warm_blooded: predicate;

        # Mammals are hairy and warm-blooded
        mammal: type is
                property hairy &amp; warm_blooded;

        # Dogs are mammals
        dog: type is
                property mammal;
```

```
# Poodles are dogs
poodle: type is
        property dog;

# Collies are long-haired, brown-haired dogs
collie: type is long_haired brown_haired dog;

# Given the above, all of the following propositions are true
confirm any poodle isa hairy all;
confirm brown_haired dog << mammal;
confirm any collie isa warm_blooded all;
```

Note that parameterized types such as hairy(t: type) allow subtypes to be defined in the same way that adjectives qualify nouns in English to select subtypes (e.g., hairy dogs). This use of adjectives provides Easel with a powerful mechanism for defining anonymous subtypes "on the fly." In a similar way, the use of quantifiers such as **any**, **all**, and **some** provides Easel with a means of selecting subtypes that have to be manipulated or tested. These concepts and constructs are elaborated on in subsequent chapters.

2.5.1. lambda(list type, rt: type): type;

Any definition may be parameterized. **Lambda** is the type of all parameterized types and operations. The first parameter to lambda is the formal parameter list of the type or operation. The second parameter is the return type. The specification `f(x: t; y: s): u;` is a shorthand for `f: lambda([ x: t; y: s], u): type;`. The type of a parameterless operation is `lambda(nil, ?)`. Syntactically, the formal parameters of a lambda definition are surrounded by parentheses and placed between the name and semi colon of the define. The type specified after the colon in the type definition becomes the `rt` of the lambda and the `lambda` type is the second parameter to the define.

Restrictions

Only the **define** form of **lambda types** has been implemented.

2.5.2. control "_fp_spec"(identifier, type): type;

A formal parameter is a special form of **attribute**. It is an **attribute** of both the lambda subtype and of all examples of that subtype.

Formal parameters of a type are unassignable (because they are attributes of types and all types are immutable). However, if the type is mutable, it is legal to assign to the formal parameters of an example of the type. (The same is true of all attributes of types, not just formal parameters.)

A formal parameter specification gives the domain of actual parameter values for each formal parameter of a lambda type. Formal parameters may have identifier names. If a formal parameter is named, its value may be referenced by that identifier from within the formal parameter list and within the body of the lambda definition. The actual parameter value may also be referenced by dot qualification on the lambda type or any example of the lambda type.

By convention, if a lambda definition gives special treatment to a subtype of the type of a formal parameter, the formal parameter type is specified as a union of the subtype and the type.

2.5.3. "..."(any): any;

The variable number of parameters suffix is applied to the type of the last formal parameter.

Any number (i.e. zero or more) of actual parameters may correspond to a variable number of parameters formal parameter.

The actual parameters corresponding to a "..." formal parameter are collected into a list whose components may referenced by indexing the formal parameter name (see 6.7.5 and 6.8.4).

Commentary

In the sample code below, the bezier function takes an arbitrary number of points and returns a path between them of the specified color.

```
# STATUS: current

bezier(color, points: point...): path;
# ...
B :: path := bezier(red, [0,0], [1,0], [1,1], [0,1], [0,0]);
```

In the example below, we use a list parameter to write a function which returns the sum of an arbitrary number of integers.

```
# STATUS: planned

example(): action is
        # Function to compute the total of
        # an arbitrary number of integers
        my_sum(X: int...): int is

                # Initialize the total
                total :: int := 0;

                # Sum over the values of the X examples
                for i: each X do
                        total := total + i;
                return total;

        # Output their sum
        confirm sum(1, 5, 999) = 1005;

example();
```

Restrictions

The variable number of parameters operator may not currently be used in user progroms, only in Package Standard.

2.6. The Universal Type, Unspecified, and Nil

The universal type is the type of no properties and all examples. It is a supertype of all types and a subtype of only itself. Formally, it has no properties that are not tautologies.

The quantifier **all** (see 3.5.2) may be used to reference the universal type. The quantifier **any** (see 3.5.1) may be used to reference any arbitrary example of the universal type. The universal type may also be referenced, and usually is throughout this manual, as **any type**.

Commentary

The universal type is the type of no properties and all examples. It may be referenced in

Easel using the quantifier **all**".

```
# STATUS: current

confirm 3 isa all;
confirm "hello" isa all;
confirm list'[0, 1] isa all;
confirm list isa all;
confirm int isa all;
```

It includes the type "false".

```
# STATUS: current

A:: boolean := true;

confirm (A & !A) isa all;
confirm false isa all;
confirm 3 > 10 isa all;
```

It is a supertype of all types and a subtype of only itself.

```
# STATUS: current

bear: type;
confirm all >> bear;
confirm all >>int;
confirm all >> all;
confirm (! (all << bear));
confirm (! (all << int));
confirm all << all;
```

The quantifier **any** may be used to reference any arbitrary example of the universal type. The universal type may also be referenced as **any type**.

The function f shown below can be applied to any example of any easel type.

```
# STATUS: current

f(a: all): int is
        if a isa int then
                return a;
        else
                return 42;

confirm (f 38 == 38);;
confirm (f "abc" == 42);
```

Attributes of type **any** can hold values of **any** type. For example

```
# STATUS: current

amorphous :: any := 3;
confirm (amorphous + 2) = 5;

amorphous := (list string)'["x" , "y" , "z" ];
confirm amorphous[1] = "y";

amorphous := list type;
```

```
    confirm amorphous << list;

    amorphous := "my dog has fleas";
    confirm (length amorphous) = 16;
```

Here the attribute `amorphous` is created with an integer value, then re-assigned as a string value, a list of strings, and then the type `list`.

2.6.1. "?": any type;

The pseudo value, `?`, indicates that a value of an expression or action is unspecified at the point of the `?`. The unspecified value is a mechanism for incomplete and imprecise specification, for avoiding redundant specifications, and for delaying specifications. The unspecified value may be used anywhere an expression or statement is allowed. The effect of "?" depends on the context of its use. In its most general use, it is a form of incomplete or imprecise specification and indicates that the information is unknown, unavailable or unneeded. The interpretation of unspecified depends on the context of its use. As the name in an attribute definition (see 6.2.1), unspecified indicates that the attribute is anonymous. As the body of a define (see 3.2.1 and 7.1.1), it indicates that the definition is incomplete. As the initial value of an attribute (see 6.2.1), it indicates that the initial value is not specified at that point. As the value of an actual parameter to an operation or parameterized type (see), it returns a subtype of the type with the specified parameters instantiated.

Commentary

The pseudo value (?), indicates that a value of an expression or action is unspecified at the point of ?. For example,

```
    # STATUS: current

    i :: int := ?;
```

Here the value being assigned to is unspecified.

The unspecified value is a mechanism for incomplete and imprecise specification, for avoiding redundant specifications, and for delaying specifications. For example, the following statements create a new triangle attribute whose color is unspecified - that is, its subtype is (3, ?). Later that attribute is assigned two values, one with green and the other with yellow.

```
    # STATUS: planned

    polygon(sides: int, c: pattern): type is
            x:: int := sides;
    triangle :: polygon(3, ?);
    triangle := new polygon(3, green);
    triangle := new polygon(3, yellow);
```

The unspecified value may be used anywhere an expression or statement is allowed, although a runtime error will be raised if an attribute whose value is unspecified is referenced, as shown in the examples below:

```
    # STATUS: current

    theta:: Angle := pi / 2;
    x:: int := 4;
```

```
y:: int := ?;
z:: int := 2;

if y == y then # Error - attribute whose value is "?" mustn't be referenced
        outln "y is unspecified";
else
        outln "y is specified";

if z == ? then # No error - z's value is specified
        outln "y is unspecified";
else
        outln "y is specified";

theta := ?; # No error
if x = 4 then
        null ?;  # No error
```

The effect of **?** depends on the context of its use. In its most general use, it is a form of incomplete or imprecise specification and indicates that the information is unknown, unavailable, or unneeded. As the name in a **type** definition (see 3.2.1) or **attribute** definition (see 6.2.1), unspecified indicates that the definition or attribute is anonymous.

```
# STATUS: current

?: int := 3;
outln self[6]; # Output first attribute of an actor
```

As the body of a definition (see 3.1 and 8), **?** indicates that the definition is incomplete:

```
# STATUS: current

f(int): int is ?;
```

This is equivalent to:

```
# STATUS: current

f(int): int;
```

As the initial value of an attribute (see 6.2.1), **?** indicates that the initial value is not specified at that point:

```
# STATUS: current

x :: int := ?;
```

As a statement (see), **?** indicates that there may be intervening unspecified statements:

```
# STATUS: planned

if 2 + 1 = 3 then
        output "Hello";
        ?;
        output "finished the x = 3 branch";
```

Note that attempting to execute an unspecified statement will cause an error.

2.6.2. nil: any type is ?;

Nil is a pseudo value of any mutable type. It indicates an error, inconsistency, or absence of a value. It may be used anywhere that an expression is required. Nil may be assigned to and referenced from any **attribute**, passed as a parameter, or returned as the value of an operation. If an operation that expects a mutable parameter is applied to nil, and that operation has no obvious semantics when applied to nil, then the operation will return nil without reporting an error.

Note that although nil is a value of any mutable type, it may be passed to routines that otherwise expect immutables provided that the formal parameter type is a union with nil; for example "p(my_immutable_type | nil): action;".

Commentary

Nil is a pseudo value of any type and indicates an error, or the inconsistency or absence of a value. It may be used anywhere that an **expression** is required; in particular, it may be assigned to and referenced from any attribute, passed as a parameter, or returned as the value of an operation.

The following three examples show nil being assigned to an attribute, passed as a parameter, and returned from an operation:

```
# STATUS: current

nil_test(): action is
        k :: list := nil;

        v(y : int, z: int): int is
                if y = 0 then
                        return nil;
                else if z = nil then
                        return nil;
                else
                        return y / z;

        confirm k = nil;

        confirm v(6, 2) = 3;
        confirm v(6, nil) = nil;

nil_test();
```

If an error or inconsistency occurs during the evaluation of an operation with a nil parameter, the error is not reported; instead, nil is returned as the value of the operation. This allows programs to continue execution without cascading error reports. For example, in the following code the function f returns nil if it is passed a negative value, and the function g checks for a negative value returned by f before continuing:

```
# STATUS: current

f(i: int): int is
        if i < 0 then
                return nil;
        else
                return sqrt i;

g(i: int): int is
        if (f i) = nil then
                return nil;
        else
                return 3 ^ (f i);
```

```
confirm (f ~3) = nil;
confirm (g 4) = 9;
```

If a routine accepts nil as a valid actual parameter value, nil must be included in the formal parameter type. For example, in the following statements, the call on `p1` is illegal, since the formal parameter type of `p1` does not contain nil:

```
# STATUS: current

p1(int): int is
        return 93;
p2(int | nil): int is
        return 94;

outln p1 nil;
outln p2 nil;
```

(However, the current version of Easel does not detect this error.)

2.7. Operations on Types

Most types do not have names. One way to create anonymous types is to compute them using type-valued operations.

Commentary

An anonymous type is a type that is created "on the fly" to capture a special set of properties. An anonymous type is defined inline at the point of its use.

Anonymous types may be used anywhere named types can appear - for example, in property lists (A), in attribute definitions (B), in formal parameter lists (C), as the result type in function definitions (D), in iterators (E), and in select statements (F).

```
# STATUS: planned

# A
Kodiak: type is
        property small brown Alaskan bear;

# B
Fido: (faithful male tiny dog) := ?;

# C
cuteness(small brown pet bear): int;

# D
winner(football pool): young lucky male person;

# E
for every green four-sided regular polygon do
        null();

# F
select Harry of
        case elegible young bachelor then
                null ();
```

Here `small brown Alaskan bear`, `faithful male tiny dog` , `small brown pet bear` , `young lucky male person` , `green four-sided regular` polygon, and eligible young bachelor are anonymous types.

A fuller us of anonymous types is given below. Here, snakes are defined "on the fly" as legless reptiles.

```
# STATUS: planned

anonymous_example(): action is
        legless: predicate is
                has(x, num_legs) & x.num_legs = 0;
        reptile(num_legs: int): type;
        reptiles: list reptile := [new reptile 4, new reptile 0];
        for r: every reptiles do
                select r of
                        case legless reptile then
                                output "Found a snake\cr";
anonymous_example();
```

As discussed in 2.1, many types do not have names: they are created "on the fly" and called anonymous. Type-valued operations are often used to create such anonymous types. In 1.9, the specialization operator (&) was used in the example. This section formally defines the specialization operator. In addition, we define operators for **abstraction (~)**, **union (|)**, **unary abstraction (~)**, **complement (|)**, and **difference (diff)**. Note that the **|** and **~** operators have both unary and binary forms.

We also define boolean predicates to test for supertypes (>>), subtypes (<<), examples (isa), and attribution (has).

2.7.1. control "&"(type, type): type;

The **&** operation, read as "and", specializes a type to the properties of another type. This is a commutative and associative operation. The resulting type is the union of their properties and the intersection of their examples. The result is the least restrictive common subtype of the arguments. Any operation that applies to examples of either of the argument types also applies to examples of the result type.

Commentary

In the following line, the expression mammal **&** brown returns the type that has all the properties of mammals and of brown things.

```
# STATUS: current

bear: type is property mammal & brown;
```

Specialization is a commutative and associative operation, so that the following lines all return the same type.

```
# STATUS: current

mammal & brown & Alaskan;
mammal & Alaskan & brown;
(brown & Alaskan) & mammal;
brown & (Alaskan & mammal);
```

The result of specialization is the least restrictive common subtype of the arguments. For example, there are many common subtypes of mammal & brown:

```
# STATUS: current

mammal & brown & hungry;
mammal & brown & four-toed;
mammal & brown & carnivorous;
```

However, what `mammal & brown` returns is the least restrictive of them: intuitively, `mammal & brown` - i.e. all brown mammals irrespective of any other properties.

Any operation that applies to examples of either of the argument types also applies to examples of the result type. For example:

```
# STATUS: planned

Brown: type is
        french_name:: string := "maron";
mammal: type is
        french_name:: string := "mamale";

weight(mammal): int is
        return 100;
luminance(Brown): number is
        return 200;

Kodiak: type is
        property Brown & mammal;

Bruin:: Kodiak := new Kodiak;
output luminance Bruin;
output weight Bruin;
```

In this example, `luminance,` and `weight` may both be applied to `Bruin` because `Bruin` is both a `mammal` and `brown`.

Restrictions

In the current implementation, overload resolution fails on specializations used as actual parameters.

2.7.2. "~"(type, type): type;

The **abstraction** operation returns the intersection of the properties of its arguments. Abstraction is commutative and associative. The abstraction of two types is a supertype of their union type. Any operation that applies to examples of the result type also applies to examples of both argument types.

Commentary

Given two types, the binary abstraction operator returns their common supertype, i.e. the intersection their properties. For example, given

```
# STATUS: planned

bear: type;
Brown: type;
White: type;
Alaskan: type;
continental: type;

polar_bear: type is
        property bear & White;
```

```
brown_bear: type is
        property bear & Brown;
Kodiak: type is
        property brown_bear & Alaskan;
grizzly: type is
        property brown_bear & continental;

outln grizzly ~ Kodiak;
confirm (grizzly ~ Kodiak) = brown_bear;
```

Similarly, the following statements hold:

```
# STATUS: planned

confirm brown_bear ~ polar_bear = bear;
confirm grizzly ~ polar_bear = bear;
confirm polar_bear ~ white = white;
confirm grizzly ~ white = all;
```

Note that in the last example, **all** is being used to represent the empty set, i.e. the type that has no properties whatsoever and therefore describes all examples.

Abstraction is a commutative and associative operation. Thus, the following expressions all return the same type, namely bear.

```
# STATUS: planned

grizzly ~ Kodiak ~ polar_bear;
polar_bear ~ Kodiak ~ grizzly;
polar_bear ~ (Kodiak ~ grizzly);
(polar_bear ~ Kodiak) ~ grizzly;
```

As shown in the above examples, the abstraction of two types is a supertype of their union type - for example, all grizzlies are examples of `grizzly ~ Kodiak`.

Any operation that applies to examples of the result type also applies to examples of both argument types. Thus given

```
# STATUS: planned

habitat(grizzly ~ Kodiak): biome;
Bruin: grizzly;
Pearl: Kodiak;
```

it is legal to call `habitat` on both grizzlies and Kodiaks.

```
# STATUS: planned

output habitat Bruin;
output habitat Pearl;
```

2.7.3. control "|"(type, type): type;

The **type union** operation, read as **or**, returns the type which is the union of the examples of its arguments. This is a commutative and associative operation. The result is a supertype of the arguments, but its examples do not necessarily have any shared properties. Any operation that applies to examples of the result type also applies to examples of both argument types.

Commentary

The union operation returns the type which is the union of the examples of its arguments. For example:

```
# STATUS: planned

bear: predicate;
grizzly: type is property bear & brown;
polar_bear: type is property bear & white;
Kodiak: type is property bear & brown;
polygon: type;
arctic_bear: type is property Kodiak | polar_bear;
funny: type is property Kodiak | polygon;
K :: Kodiak := ?;
P :: polygon := ?;

confirm K isa funny;
confirm K isa arctic_bear;
confirm P isa funny;
```

Note that because `arctic bears` are a union of `Kodiaks` and `polar bears`, the `Kodiak K` is an `arctic bear`. (For the isa operator, see 2.7.8).

Union is a commutative and associative operation, so that the following lines are all equivalent:

```
Kodiak | polygon | polar_bear;
Kodiak | polar_bear | polygon;
(Kodiak | polar_bear) | polygon;
Kodiak | (polar_bear | polygon);
```

The result of type union is the supertype of the arguments, but its examples do not necessarily have any shared properties:

```
# STATUS: planned

confirm funny >> Kodiak;
confirm funny >> polygon;
confirm arctic_bear >> polar_bear;
confirm arctic_bear >> Kodiak;
```

Notice that even though `funny` is a supertype of both `Kodiaks` and `polygons`, `Kodiaks` and `polygons` do not share any properties. (For the supertype operator **>>**, see 2.3.1.)

Any operation that applies to examples of the result type also applies to examples of both argument types. For example:

```
# STATUS: planned

size(F: funny): int is
        select F of
                case polygon then
                        return 0;
                case Kodiak then
                        return 1;
K: Kodiak;
P: polygon;

confirm size K = 1;
confirm size P = 0;
```

The next example illustrates the **&**, **|** and **~** operators, defined above:

```
# STATUS: planned

components: type is
        # define vehicle properties
        ship: type is
                property has motor
                        & has drive_shaft
                        & has propellor
                        & has hull;
        car: type is
                property has motor
                        & has drive_shaft
                        & has wheels
                        & has chasis;
        vehicle: type is property ship | car;
        motored: type is property ship ~ car;
```

2.7.4. "~"(type): type;

The **unary abstraction** operation returns the intersection of the properties of all of the examples of its argument. Thus, for any two types t1 and t2, (t1~t2) =~(t1|t2). Unary abstraction is also useful for obtaining the type of an example.

Commentary

The **unary abstraction** operation returns the intersection of the properties of all of the examples of its argument. For example:

```
# STATUS: planned

ursine: predicate;
brownish: predicate;
whitish: predicate;
bear: type is property ursine;
grizzly: type is property bear & brownish;
polar_bear: type is property bear & whitish;
Kodiak: type is property bear & brownish;
polygonal: predicate;
polygon: type is property polygonal;
arctic_bear: type is property Kodiak | polar_bear;
funny: type is property Kodiak | polygon;

~bear = property ursine;            # tautology
~grizzly = property ursine & brownish;
~arctic_bear = property ursine;
~funny = all;
~polygon = property polygonal;
~polar_bear = property ursine & whitish;
~Kodiak = property ursine & brownish;
```

For any two types `t1` and `t2`, `(t1~t2) =~(t1|t2)` For example,

```
Kodiak ~ polar_bear = ~(Kodiak | polar_bear) = bear;
polygon ~ bear = ~(polygon | bear) = all;
```

Unary abstraction is also useful for obtaining the type of an example. To illustrate:

```
# STATUS: planned

Bruin: bear & whitish := ?;
Cody: ~Bruin := ?;
```

Here `Cody` is declared to have all the properties that `Bruin` has .

Restrictions

Unary abstraction is not processed correctly by the **isa**, **subtype**, and **supertype** operators.

2.7.5. "l"(type): type;

The **type complement** operation, read as **not**, returns the type of all examples which are not examples of its argument. The specialization of the argument with the result is the inconsistent type. The abstraction of the argument with the result is the universal type. The union of the argument with the result is also the universal type.

Commentary

The complement operation returns the type of all examples that are not examples of its argument. For example:

```
# STATUS: planned

ursine: predicate;
polygonal: predicate;
brownish: predicate;
whitish: predicate;
polygon: type is closed property polygonal;
bear: type is closed property ursine;
grizzly: type is closed property ursine & brownish;
polar_bear: type is closed property ursine & whitish;
Kodiak: type is closed property ursine & brownish;

confirm polygon << |bear;
confirm bear << |polygon;
confirm ! (bear << |polar_bear);
confirm ! (polar_bear << |bear);
```

The specialization of the argument with the result is the inconsistent type; for example:

```
# STATUS: planned

confirm ! (bear & |bear);
```

The abstraction of the argument with the result is the universal type; for example:

```
# STATUS: planned

confirm bear ~ |bear = all;
```

The union of the argument with the result is also the universal type; for example:

```
# STATUS: planned

bear | |bear = all;
```

Some further examples of the complement operator include the following:

```
# STATUS: planned

complement_example1: type is
        # Negative ints are less than zero
        negative: type is
                property int & < 0;

        # Nonnegtive ints are not negative
        non_negative: type is |negative;

        # Define some birds
        bird: type is enum (duck, swan, grouse, chicken, partridge);

        # Any bird that is not a chicken is of type non_chicken
        non_chicken: type is bird & |chicken;
```

Restrictions

The **type complement** operator is not processed correctly by the **isa**, **subtype**, and **supertype** operators.

2.7.6. "<<"(all, all): boolean;

The subtype predicate returns true iff the first argument is a subtype of the second. If neither is a subtype of the other it returns false.

Commentary

In Easel, one type is a subtype of a second if and only if the properties in the first type are a (possibly non-strict) subset of the properties of the second. For example, the result of a specialization operation is a subtype of the operands.

The subtype relationship between two types can be computed by the subtype operator **<<**, which returns true if the first argument is a subtype of the second. Thus:

```
# STATUS: current

vehicle: type;
car: type is
        property vehicle;
confirm car &lt;&lt; vehicle;


# STATUS: planned

ursine: predicate;
mammal: predicate;
polygonal: predicate;
brownish: predicate;
whitish: predicate;
Alaskan: predicate;
polygon: type is closed property polygonal;
bear: type is closed property ursine & mammal;
grizzly: type is closed property ursine & brownish;
polar_bear: type is closed property ursine & whitish;
Kodiak: type is closed property bear & brownish & Alaskan;

confirm Kodiak << bear;
confirm Kodiak << Alaskan all;
confirm Kodiak << mammal all;
confirm grizzly << brownish all;
confirm bear << mammal all;
```

```
    confirm ! Alaskan all << brownish all;
    confirm ! brownish all << bear;
```

2.7.7. ">>"(all, all): boolean;

The **supertype predicate** operation returns true iff the first argument is a supertype of the second. If neither is a supertype of the other it returns false.

Commentary

In Easel, one type is the supertype of a second if and only if the properties of the second are a subset of the properties of the first. The supertype relationship can be computed using the supertype operator **>>**, which returns true if the first argument is a supertype of the second.

The following is a simple example of the supertype predicate:

```
    # STATUS: current

    animal: type is
            mass :: number := ?;

    dog: type is
            property animal;

    confirm animal >> dog;
```

2.7.8. "isa"(all, type | list): boolean;

The **isa** operation returns true iff the first argument is an example of the second.

Commentary

Any entity in Easel is said to be an example of a type if and only if it exhibits all of the properties included in that type. In the following very simple case, it is confirmed that `compaq` is an example of the type `computer`.

```
    # STATUS: current

    computer: mutable type is
            price: number := 1500.00;
    compaq :: computer := new computer;
    confirm compaq isa computer;
```

In the slightly more complex example below, `i` is an example of the type `even`, because it is an integer and is evenly divisible by 2.

```
    # STATUS: planned

    even: type is
            property int & mod(it, 2) = 0;
    i :: int := 4;
    confirm i isa even;
```

The example of relationship can be computed by the **isa** operator, which returns true if its first argument is an example of its second argument. Thus in the example above, we can write

```
    # STATUS: planned
```

```
confirm i isa even;
```

Given the above:

```
# STATUS: planned

Peter :: bear := ?;
Sam :: Kodiak := ?

confirm Peter isa bear;
confirm Peter isa brownish mammal all;
confirm Peter isa mammal all;
confirm Sam isa Kodiak;
confirm Sam isa brownish all;
confirm Sam isa brownish Kodiak;
confirm ! Peter isa Kodiak;
confirm Peter isa |Kodiak;
```

2.7.9. "( )"((any type)...): the type;

Expressions must sometimes be enclosed in parentheses to override the operator bindings built into the language, to enclose aggregates as arguments to single argument lambdas, or for clarity. Parentheses are interpreted as an identity function on examples of any type.

Commentary

Expressions must sometimes be enclosed in parentheses to override the operator bindings built into the language or be used otherwise for clarity. Parentheses are interpreted as an identity function on examples of any type.

```
# STATUS: current

confirm 3+5*7=38;
confirm (3+5)*7=56;
confirm (3)=3;
```

Functions in Easel have relatively low binding strength, so that the following holds:

```
# STATUS: current

f(i: int): int is
        return 2 * i;

confirm (f 3 + 5) = 16;
```

Parenthesizing the first parameter to `f` does not change the association, because `f  3` and `f(3)` are identical:

```
# STATUS: current

f(i: int): int is
        return (i * 2);

confirm (f (3) + 5) = 16;
```

To force `f` to be applied to just 3, the entire function call must be parenthesized:

```
# STATUS: current

f(i: int) : int is
        return (i * 2);

confirm ((f 3) + 5) = 11;
```

2.7.10. "="(any type, any type): boolean;

The **equality** operation returns true iff its arguments are references to the same example. Equality may be applied to (references to) any two examples regardless of their types.

Commentary

The **equality** operation returns true if its arguments are references to the same example. The **inequality** operation returns true if and only if its arguments are not references to the same example. For example in the following statements, `f` and `g` are both references to the value 3.0 of the immutable type number, while `h` references the value 4; and `carnation` and `rose` are references to two distinct examples of `red flowers`.

```
# STATUS: current

f :: number := 3.0;
g :: number := 3.0;
h :: number := 4.0;
confirm f = g;
confirm f != h;

flower(C: pattern): type is
        petals :: int := 4;
carnation :: flower := new flower red;
rose :: flower := new flower red;
confirm rose != carnation;
```

As shown below, equality and inequality may be applied to (or refer to) any two examples regardless of their types:

```
# STATUS: current

confirm rose != g;
confirm Peter != 3;
```

2.7.11. "=="(any type, any type): boolean;

The **structural equality** operation returns true iff all of its arguments' attributes have the same value or structure. Structural equality will return true if applied to two mutables with the same attribute values.

Commentary

The **==** operation returns true even if its arguments are different examples, provided they both have the same value or structure. This is different from **=**, which returns true only if its arguments are references to the same example. (Note that **=** and **==** are equivalent for immutables.)

The following example illustrates the difference between **=** and **==**.

```
# STATUS: current


L1: list int := list'[99, 100, 101];
L2: list int := L1;

L3: list int := new list int;
L4: list int := new list int;

L3 := (list int)'[99, 100, 101];
L4 := (list int)'[99, 100, 101];

confirm L1 = L2;
confirm L1 == L2;
confirm L3 != L4;
confirm L3 == L4;
```

2.7.12. "!="(x: any type, y: any type): boolean #{is !x = y}#;

The **inequality** operation returns true iff its arguments are not references to the same example. Inequality may be applied to any two examples regardless of their types.

2.7.13. "!=="(any type, any type): boolean;

The **structural inequality** operation returns the negaton of the structural equality operation.

2.8. Booleans

2.8.1. boolean: type #{ is type }#;

Boolean is a type with five truth values: true, probable, maybe, unlikely, and false. Truth is the property of being provably in accordance with some belief system. Any type can be viewed as a belief system in which the properties are the beliefs. Thus, to ask whether a boolean expression is true at some point in a program, is to ask whether it expresses a property that is consistent with the current type or belief system at that point.

Probable means that all the available evidence is consistent with the belief system. Maybe is an exceptional condition meaning there is no way to determine the consistency or inconsistency of the property. Unlikely means that all the available evidence is inconsistent with the belief system, while false is a truth value that is inconsistent with the current belief system.

2.8.2. true: boolean;

True is a truth value that is provably consistent with the current belief system. That is, it is the type of all consistent types.

2.8.3. probable: boolean;

Probable is a truth value that as far as can be determined is consistent with the current belief system. That is, it is the type of all types consistent with the known properties of the current belief system.

2.8.4. maybe: boolean;

Maybe is a truth value whose consistency with the current belief system cannot be determined. That is, it is the type of all types that are neither consistent or inconsistent with the current belief system.

2.8.5. unlikely: boolean;

Unlikely is a truth value that as far as can be determined is not consistent with the current belief system. That is, it is the type of all types inconsistent with the known properties of the current belief system.

2.8.6. false: boolean;

False is a truth value that is inconsistent with the current belief system. That, it is the type of all inconsistent types.

2.9. Operations on Booleans

2.9.1. "!"(type): boolean;

Because true and false are types they inherit all of the operations defined on types (see 2.7 and 2.13). When the binary type operations **&** and **|**are applied to types the produce results with the intended boolean interpretation. **Unary |** however yields the correct boolean interpretation however only when applied to true or false. Thus, the boolean not operation **!** is provided. When applied to any consistent type including true, false is returned. When applied to an inconsistent type, not returns true.

2.9.2. not(type): boolean;

The "not" operator is a synonym for the unary "!" operator.

2.10. Dimensional Analysis

A dimension is any fundamental property such as mass, length or time on which measurement is possible. Dimensional analysis is a process for verifying the consistency of the dimensions used in numeric computations, such that for example only length squared and not distance may be used as a measure of area. The author must specify the dimensionality of each formal parameter and of the return type of each operation. The translator will verify that each actual parameter in an application has the required dimensionality and that the operation produces a result of the specified dimensionality. Note that many mathematical functions are dimensionless.

Restrictions

Dimensional types may be declared and referenced, but no correctness checking is performed. In addition, implicit multiplication of units is unimplemented: one must write "3 * km", not "3 km".

2.10.1. dimension: number type;

Dimension is the type of all dimensions. The body of a dimension may be used to specify the relationships among dimensions.

The SI base dimensions of `Length` and `Time` are defined in this section, along with the supplemental dimension `Angle`. Other SI base dimensions, the SI base units, and some frequently-used derived dimensions and units are defined in the Metrics Library. Note that to avoid naming conflicts, the names of dimensions are capitalized in Easel.

Commentary

Dimension is the type of all dimensions and can be declared like other types. Relations between dimensions can be specified in their bodies. For example in the following statements,

```
# STATUS: current

money: dimension;
capita: dimension;
wealth: dimension is money / capita;
area: dimension is Length * Length;
```

`money` and `capita` are defined as base dimensions, while `wealth`, and `area` are defined as derived dimensions.

Restrictions

Although dimensions and units may be declared and used, no dimensional analysis is performed to ensure the correctness of user programs.

2.10.2. Length: dimension;

Length is a built-in dimension in Easel.

2.10.3. Angle: dimension;

Angle is a built-in dimension in Easel.

2.10.4. Time: dimension;

Time is a built-in dimension in Easel.

2.11. Units of Measure

A unit of measure is a standard for measuring within a dimension. Each dimension must have at least one unit of measure and may have several for variations in scale or preferences for particular measuring systems. Measuring systems may be systematic (e.g. the metric system) or ad hoc (e.g., the English system).

2.11.1. unit(dimension): dimension;

Each unit of measure is an example of the subtype of unit for its dimension. Unless otherwise specified, it is assumed that all measures are scalable from zero. That is, two meters are twice the distance of one meter.

The system automatically assigns a constant to each unit of measure. The exact value of the constant is unimportant as long is the zero value corresponds to the origin of the dimension and that the relative values of different units are maintained. In practice, the values or normally assigned to match the first unit defined for a dimension.

Commentary

A unit of measure is a standard for measuring within a dimension; for example, `meter` is a unit within the dimension `length` . Each referenced dimension must have at least one unit of measure and may have several for variations in scale or preferences for particular measuring systems. Measuring systems may be systematic (e.g., the metric system) or ad hoc (e.g., the English system).

Each unit of measure is an example of the subtype of the type unit for its dimension. For example, the statements below declare dollar to be an example of unit money, which is a subtype of unit specialized for money.

```
# STATUS: current

money: dimension;
dollars: unit money is 1;
yen: (unit money) := dollars/100;
meters: unit Length;
```

Units are associated with dimensional constants, but not with attributes whose types are dimensional. Thus in the following code the attributes `salary` and `length` are of type `money` and `dollar` , respectively, but are not measured in any particular unit. The expressions `4000 dollar` and `2 meters` , however, are measured in the units `dollar` and `meter`, respectively.

```
# STATUS: current

salary := 4000 dollars;
Length := 2 meters;
```

Units are not needed when referencing dimensioned attributes; all necessary conversions are done internally:

```
# STATUS: current

US_invoice :: money := 10 dollars;
Japanese_invoice :: money := 1000 yen;
confirm Japanese_invoice == US_invoice; # No units necessary
```

2.12. Outputting Dimensioned Values

Dimensioned values may be converted to any desired unit by dividing by that desired unit.

Commentary

Internally, dimensioned values are maintained in a canonical unit, usually the first unit of the dimension to be declared. To output dimensioned values, expressions need to be converted to the desired units using the / operator. For example in the following statements, the canonical unit is meters; when `p` is declared, its value is set to 3000000 `meters`:

```
# STATUS: planned

m: unit Length;
km: unit Length is 1000 m;
Mm: unit Length is 1000000 m;
p :: Length := 3 Mm;
output("p is ", p / km, " km\cr");
```

When the output statement is executed, `p` in `kilometers` will evaluate to 3000000 / 1000 or 3000, so the statement will print the following:

```
# STATUS: planned

p is 3000 kilometers
```

When the conversion involves units with offsets, the / operator takes the offset into account.

```
# STATUS: planned

temperature: dimension;
K: unit temperature;
deg_Celsius: unit(temperature, 273.15);
T: temperature := 20 deg_Celsius;
output (T / K, "  K"); # Prints 293.15 K;
```

2.13. Representation Types

All representations of programs and data are currently determined by the translator and interpreter and cannot be specified by authors. The system is however capable of supporting multiple representations for a given type. In fact, examples of type int are sometimes represented as 16 bit integers and sometimes as floating point with implicit conversions between them as needed. Subtypes with the same attributes but different allocations of attributes to formal parameters have multiple representations with implicit conversions among them.

2.14. Representation Specifications

Each different representation for examples of a type is a representational subtype of that type. The choice multiple equivalent representations does will not alter the outcome of any correct computation, but may affect the performance. No mechanisms are currently provided for explicit specification or author control of representations.

Chapter 3. Examples

An example of a type is anything that satisfies or conforms to all properties of that type. Thus, examples are not restricted to things that can exist or be accurately represented in a computer. Instead, examples are restricted to those things whose types can be accurately described or represented in a computer. Easel makes no attempt to represent examples, but instead provides extensive facilities for describing types and uses quantification to reference examples of any named or described type. Each reference refers to, means, or denotes a example, but does not encode the example.

An anaphoric reference is any grammatical form used to reference a type or example without use of its proper name. Such mechanisms include examples referenced by pronouns (3.4) and quantifiers (3.5), and types referenced using adjectives (3.11), type valued computations (2.7) and other forms of anonymous types.

3.1. Defining and Referencing Examples

Proper nouns are the names of examples. They include language defined literals, author defined literals, the names of examples of enumeration types, and any defined identifier. The visibility rules for proper nouns are the same whether a literal, type, operation or other example is named. All definitions define an example of some type.

Commentary

An example of a type is anything that satisfies or conforms to all properties of that type. Thus, examples are not restricted to things that can exist or be accurately represented in a computer. Instead, examples are restricted to those things whose types can be described or represented accurately in a computer. Easel makes no attempt to represent examples, but instead provides extensive facilities for describing types and uses quantification to reference examples of any named or described type. Each reference refers to, means, or denotes a example, but does not encode the example. Thus, we may reference a cigar inside the computer, but that representation is never complete - it can never express all the properties of the physical cigar. However, it may approximately represent these properties that are important to the simulation - weight, color and so on. Thus when we generate a new example in the simulation, this is really a reference to an actual or imagined cigar.

Proper nouns are the names of examples. They include language-defined literals, author-defined literals, the names of examples of enumeration types, and any defined identifier. The visibility rules for proper nouns are the same whether a literal, type, operation, or other example is named. All definitions define an example of some type.

Easel makes frequent use of anaphoric references; such references are frequently in natural language. English examples include the following:

```
John threw the ball. It broke the window.
Mary threw the ball. She broke her arm.
Tom tried to open the lunch box. Its handle came off.
```

The context of the reference provides sufficient implicit information that makes the statement meaningful.

More formally, an anaphoric reference is any grammatical form used to reference a type or example without use of its proper name. Easel uses such limited forms of anaphoric references to allow the manipulation of types and examples. In Chapter 2, we used the it construct extensively. Easel mechanisms include examples referenced by pronouns (3.4) and quantifiers (3.5), and types referenced using adjectives (3.11), type-valued computations (2.7), and other forms of anonymous types.

3.2. Generating Examples

Commentary

Types can be of two varieties: immutable and mutable. Examples of immutable types do not change their values throughout the program execution, while mutable types can have examples whose properties can change.

In the following statements, dog is mutable, since it has properties that can vary. In order to allocate a representation (storage) for the example, the operator **new** is used.

```
# STATUS: current

dog(height: int): type is
        age :: int := ?;
Fido :: dog := new dog 10;
Fido.age := 5;
confirm Fido.age = 5;
```

However, one cannot reference the attributes of a type nor an example of a type that has not had space allocated, as in this example:

```
# STATUS: current

dog: type is
        age :: int := 5;
Fido: dog;
confirm dog.age = 5;  # error
confirm Fido.age = 5;  # error
```

Thus **new** creates a reference to an example of a mutable type. Each new reference is unique and distinguishable from all other references. Multiple references may however refer to the same example.

As indicated above, examples can be tagged with proper names. However, it is sometimes useful to have a collection of examples that are not all named individually. These are called anonymous examples. An anonymous example can be defined using an expression such as `null new dog`. The following example illustrates how an anonymous example can be used.

```
# STATUS: current

new_example(): action is
        # Define the shelf type as holding 20 items
        shelf(id: int): type is
                num_items :: int := rand(0, 20);
        pantry: (list shelf) := new list shelf;
        for i: each (1..100) do
                push(pantry, new shelf i);      # generates 100 shelves
        for s: each pantry do
                # shelves with less that 10 items are restocked
                if s.num_items < 20 then
```

```
                            output("Restocking shelf ", s.id, ", which only has ",
                                    s.num_items, " items.\cr");
                            s.num_items := 20;
    new_example();
```

3.2.1. control "_define"(identifier, t: type, t): property;

The **define** operation associates a name with an expression of any type. The first parameter is the name being defined and must be an identifier. The second parameter is any type of the value of the expression. The third parameter is an expression for the value.

The name will be visible throughout the most local enclosing simulation scope of the definition, except where hidden by a more local attribute of the same name. The type must be some type of every value of the expression. The expression is evaluated each time the name is referenced and the resulting value returned as the value of the reference.

If the first argument is unspecified, the definition is anonymous. If the second argument is unspecified, there are no restrictions on the type of the result and is similar to specifying the result type as all. If the third argument is unspecified, the definition is incomplete.

Commentary

```
    # STATUS: current

    t1: type;
    p1: number;
    f1(int): number;

    t2: type is ?;
    p2: number is ?;
    f2(int): number is ?;

    t3: type is int;
    p3: number is 3.197;
    f3(i: int): number is return i + 10;
```

Here `t1`, `p1`, and `f1` are definitions without bodies, declaring a type, a numeric constant, and a function, respectively. They are equivalent to `t2`, `p2`, and `f2`, which provide an explicitly unspecified body. Finally, `t3`, `p3`, and `f3` provide expressions which complete the definitions.

Defines are readily distinguished from attribute definitions (see 6.2.1) by the fact that attribute definitions use a double colon:

```
    # STATUS: current

    p1: number;
    p2 :: number;
```

Here `p1` is a numeric constant with an unspecified value, while `p2` is a numeric attribute with an unspecified initial value.

3.3. Visibility of References

Defined names are visible throughout their most local enclosing simulation. That is, they are defined in a scope that is global to the entire simulation, regardless of where their definition appears within the simulation, and thus are called global definitions. If several global definitions have the same name, all are visible, and no conflict occurs if they have different signatures (i.e. name, number and type of parameters, and return type). If two global defines have the same or overlapping signatures, they are legal iff they have the same value or affect in their overlap region.

It is not possible to globally define attributes in Easel, so no conflicts can arise. An attribute definition however does hide all more global defines of the same name, but only for the immediate scope of the attribute definition and in particular not for subscopes of the attribute definition. If, however, a references is made to the name of a local attribute but the specified type of the attribute is illegal for the context of the reference, then global defines of the name are visible.

3.3.1. "ref"(token): any type;

The **define reference** operation returns all definitions of the name that are visible at the point of reference. Multiple simultaneously visible definitions of a name is common. The body of each definition defines a type. The reference operation returns the union of those types.

Each type in the union is called a candidate. Subsequent processing may reduce the number of candidates in the union type, either by formal parameter resolution, context resolution, or optimization selection (3.9.1).

If no definitions of the name are visible, an error is reported and the token itself is returned. The visibility rules guarantee that attributes and defines of the same name are never simultaneously visible. Syntactically, the appearance of the identifier name alone constitutes a reference.

3.4. Pronouns

A pronoun is a language-defined identifier that can substitute for a noun or noun phrase when the referent can be understood from the context of the reference. Pronouns have global visibility but local meaning.

Additional pronouns named `self` and `sim` are defined in 8.1.4 and 8.3.17.

3.4.1. pronoun(property): property;

Pronoun specifications apply only to definitions. They make the definition visible at the language level. Each pronoun typically has other built in significance.

Restrictions

User-defined pronouns have not been implemented. They will **parse** and **sem** correctly but there is no semantics attached to them.

3.5. Quantifiers

A quantifier is a language-defined identifier that names an operation that returns a reference to an example given a type. Because quantifiers are syntactically more binding than operators, they are convenient for marking the beginning of actual parameters without the use of parentheses, especially when the quantification is applied to an anonymous type composed from adjectives.

Commentary

Consider the problem of calling the subroutine `locate` on any large available nearby `ambulance`. With traditional syntax, we would have to write:

```
# STATUS: current

locate(any(large(nearby(ambulance))));
```

Using quantifier syntax, we can simply write:

```
# STATUS: current

locate any large nearby ambulance;
```

3.5.1. any(t: type): any;

The **any** quantifier returns a reference to an arbitrary example of its argument type. It makes a worst case selection, if possible referencing an example that will later lead to an inconsistency. An error will be reported later if any example leads to an inconsistency. The **any** quantifier should be used only in situations where any arbitrary example will suffice.

Commentary

The **any** quantifier returns a reference to an arbitrary example of its argument type. It makes a worst-case selection, if possible referencing an example that will later lead to an inconsistency.

```
# STATUS: planned

t: type is 10 .. 20;
e :: t := any t;

output e; # prints "any t"
if e < 0 then
        fail();
output e; # prints "any t"
if e isa list then
        fail();
output e; # prints "any t"
```

An error will be reported later if any example leads to an inconsistency.

```
# STATUS: planned

t: type is 10 .. 20;
e :: t := any t;

if e > 15 then
        fail();
output e; # prints an error message
```

Here an error is reported because there are values of `t` that are greater than 15.

The any quantifier should be used only in situations where an arbitrary example will suffice.

3.5.2. "all" (t: type): type #{is t}#;

The **all** quantifier returns a reference to the type of all of the examples of the argument type. It selects all of the examples at once and thus is an identity operation on types. Its primary use is to obtain the syntactic benefits of quantification for anonymous types.

Commentary

The **all** quantifier returns a reference to the type of all of the examples of the argument type. It selects all of the examples at once and thus is an identity operation on types. It is used primarily to obtain the syntactic benefits of quantification for anonymous types. For example:

```
# STATUS: planned

confirm all green parrot << parrot;
```

This is the same as saying

```
# STATUS: planned

confirm green parrot << parrot;
```

(Need example of "syntactic benefits").

3.5.3. quant_exp(quant: any|number, t: type): any;

An internal operator used to construct quantified expressions. "Any" as the first parameter stands for quantifiers.

3.6. Numeric Literal Quantifiers

3.6.1. control _int_quantification(any): any;

A numeric quantifier returns a reference to some examples of its argument type. It makes a best case selection of the number of examples specified, if possible referencing only examples that will not later lead to an inconsistency. That is, `8 dogs` is equivalent to `some 8 dogs`. An error will be reported if more than the specified number of examples of the argument type lead to an inconsistency. Like the all quantifier, the result of numeric quantification is generally interpreted as a type. It is common to quantify the result of a numeric quantification.

Commentary

A numeric quantifier returns a reference to some examples of its argument type. It makes a best-case selection of the number of examples specified, if possible referencing only examples that will not later lead to an inconsistency. That is, `8 dogs` is equivalent to `some 8 dogs`.

Here is an example of using numeric quantification with the **new** operator to create multiple examples of the same **actor** type:

```
# STATUS: current

multi_sim: simulation type is
        count :: int := 0;
        mults: (list personality) := ?;
personality(): actor type is
        sim.count := sim.count + 1;
        output("I am personality ", sim.count, " of ",
                length sim.mults, ".\cr");
        for every true do
                wait 1.0;
M :: multi_sim := new multi_sim;
M.mults := 5 new(M, personality());
```

Note that in this case numeric quantification returns a list.

When used in numeric expressions, numeric quantification is equivalent to multiplication:

```
# STATUS: current

i :: int := 3 15 2;
confirm i = 90;
confirm 5 7 2 6 = 420;
confirm 5 3 * 2 = 30;
confirm 2 1.5 = 3.0;
```

When used with an action, numeric quantification causes the action to be repeated the specified number of times:

```
# STATUS: current

5 output "abc "; # prints "abc abc abc abc abc"
```

An error will be reported if more than the specified number of examples of the argument type lead to an inconsistency. Like the all quantifier, the result of numeric quantification is generally interpreted as a type. It is common to quantify the result of a numeric quantification.

The following example defines a `baker's dozen` to be 13 things which have the property `roll`:

```
# STATUS: planned

bakers_dozen: type is some 13 any & it.type = roll;
```

3.7. Stand Alone Quantifiers

Any quantifier may be used alone without an argument. In such cases, the result is the same as if the quantifier had been applied to the universal type. For example, **all** is all examples of the universal type, and **any** is any example of the universal type.

Restrictions

Stand-alone quantifiers are unimplemented.

3.8. Computed Quantifiers

Numeric quantifiers may be computed. The interpretation of computed quantifiers is the same as numeric literal quantifiers. The syntactic binding of computed quantifiers however is that of function names rather than quantifiers. Often it will be necessary syntactically, to enclose the computed quantifier together with its argument in parentheses.

Commentary

```
# STATUS: current

ant_hill: simulation type is
        count: int := 0;
ant(): actor type is
        sim.count := sim.count + 1;
        wait 1.0;

i:: int := 3;
```

```
    s: ant_hill := new ant_hill;
    x :: list ant := (i+4) new(s, ant());
    wait s;
    confirm s.count = i + 4;
```

3.9. Lambda Application

A lambda application is an instantiation of lambda type or a operation with actual parameter values and evaluation of the resulting subtype or operation instance. Generally, it is necessary only to specify the name of the type or operation and actual parameter values of the correct number and type to uniquely distinguish which lambda is to be applied. The operation or lambda type however may be ambiguously referenced (3.3.1) or may be computed. The actual parameter expression may also be ambiguous. The precise disambiguation of application is given in this section.

3.9.1. "apply"(t: lambda, (any type)...): t.rt;

Application of a parameterized type to the required number of actual parameter values of the required types, returns the value resulting from executing the body of the lambda definition. The lambda type to be applied is the union of all currently visible definitions of that name. The set of candidate definitions however is reduced by the following rules:

a. Eliminate all candidates that are not lambda types,
b. Eliminate all candidates that do not accept the number of actual parameters in the application,
c. Eliminate all procedure candidates in expression contexts, and all non procedure candidates in statement contexts,
d. Eliminate every candidate with a formal parameter type that is not a type of the corresponding actual parameter value, unless that would eliminate all candidates,
e. If no candidate is valid, apply the implicit conversion rules of 3.10 to obtain a valid for reinterpreted actual parameters,
f. If no candidates remain, an error is reported and nil is returned,
g. If multiple candidates remain but would produce different values if applied to the actual parameters, there is an inconsistency in the program, an error is reported, and nil is returned
h. Otherwise one of the remaining choices is selected by the interpreter (because the choice does not affect the result, the choice will be either the most efficient or arbitrary),
i. The body of the only remaining candidate is then instantiated with the actual parameter values as the initial values of the formal parameter attributes,
j. If body is an operation (i.e. has statements) the body is evaluated and the resulting value returned,
k. If the body is a type (i.e. does not have statements), a subtype with the parameters instantiated is returned.

3.10. Implicit Conversions

An actual parameter need not always be an example of the corresponding formal parameter type. This section describes the exceptions. These rules are applied only at the times and places described in 3.9.1. There may be multiple candidates for the actual parameter. The rules are applied only when none of the candidates satisfy the corresponding formal parameter type. All but the last rule is to be independently applied to each of the actual parameter candidates:

a. If an example is used where a singleton type of which that example is the only example is allowed, replace the argument by the singleton type,
b. If a singleton type is used where its only example is allowed, replace the argument by its only example,
c. If a value of type `t` is used where a length one list of type `t` is allowed, replace the argument by a list containing only the argument,
d. If a list of length one is used where a that one element is allowed, replace the argument by its only element,
e. If a `lambda([ ], t)` is used where a t is allowed, apply the lambda,
f. If none of these rules apply eliminate the candidate,
g. If the same rule above will convert all the the actual parameter candidates, return the converted candidate, otherwise report an error and return nil.

Restrictions

Conversions `c`, `d`, and `e` have not been implemented.

3.10.1. ""(t: any, x: any): t #{is x}#;

Type qualification is used to explicitly specify the type of an expression. It can be used for disambiguation, emphasis or clarity, or to override the rules of 3.10. Type quantification can be used to differentiate among definitions of constants, enumeration elements and types of the same name. When an actual parameter is type qualified, the t parameter replaces the formal parameter type when the rules of 3.10 are applied, and any further application of the those rules to satisfy the parent's formal parameter type are inhibited. There are also some special case rules as follows:

a. If reference to a name is type qualified to **token**, the token itself is returned,
b. If an expression is type qualified to **expression**, the expression itself is returned,
c. If an aggregate is type qualified to `t`, `t` must be a composite type compatible with the aggregate,
d. If an expression is type qualified to the **universal type**, any candidate elimination based on return type is inhibited, but the conversions of 3.10 are reenabled for the entire candidate list if needed.

Restrictions

If the second parameter is an aggregate, its type is set to the **type** specified by the first parameter. Otherwise the second parameter is returned unchanged.

3.10.2. convert(t: type, x: any): t;

The convert operation returns the t equivalent of its second parameter, i.e. it converts x to type t.

Restrictions

Currently the only conversion supported is RGB color to vector.

3.11. Adjectives

Syntactically an adjective is an identifier that names a property and is used to modify a type to produce an anonymous subtype of that type. Adjectives are generally defined by authors and not built into the language. Adjectives generally return a subtype of their argument. In actually usage, a sequence of adjectives would normally be preceded by a quantifier to select an example form the type and followed by an improper noun naming a type.

3.11.1. "apply"(type, type type): type;

When the second argument to an application is the type **type** or all types, then a type may be used as the first argument to indicate the type of all subtypes of the first parameter. Thus type **type** is the metatype of all types, that is the type of which all types are examples. Enumeration type is the metatype of all enumerations types, that is the type of which every enumeration type is an example.

Commentary

```
# STATUS: current

example(): action is
        ant(): actor type is
                for each true do
                        wait 1.0;
```

Here `actor type` is a subtype of all types that is restricted to actors. Note that examples of `actor type` are not actors - they are **types** whose examples are actors.

This is most commonly used for declaring **actor** types and **simulation** types.

Restrictions

Implemented only for actors and simulation.

Chapter 4. Ordered and Scalar Types

4.1. Ordered Types

4.1.1. ordered: type;

An ordered type is any type whose examples are partially or completely ordered.

Commentary

As defined above, an ordered type is any type whose examples are partially or completely ordered. The type **ordered** is the supertype of all ordered types in Easel.

```
# STATUS: planned

olives: type is enum(large, jumbo, gigantic, colossal);
my_ints: type is 2..6;
confirm olives << ordered; # olives are a subtype of type ordered
confirm my_ints << ordered; # my_ints are a subtype of type ordered
```

Users can declare a new ordered type by defining a **less-than** operator 4.1. New types can be declared to inherit from ordered, but unless the **less-than** operator is defined, the type cannot be used in relational expressions.

```
# STATUS: planned

cars: type is ordered;
sedan: type is
        property cars;
SUV: type is
        property cars;
Honda: sedan := new car;
Trooper: SUV := new car;
if (Honda < Trooper) then
        null();# Error
```

Here the simple fact of declaring `cars` to be ordered does not define a comparison operator, so that the if condition fails. Note however that in reasoning about ordered types, the Easel system will assume that a less-than operator exists even if it has not been defined.

The type ordered is used in the definition of enumerations and numbers. Unless the less-than operator is defined, user-declared ordered types are not very useful, as the above example shows.

Note also that any type for which the less-than operator has been defined is a subtype of ordered, whether it is explicitly declared that way or not.

Equality is built in for all types, and the greater-than, greater-than-or-equal-to, and less-than-or-equal-to relations are all derived from the less-than relationship.

4.1.2. "<"(ordered, ordered): boolean;

Authors may define the less than operation for any ordered type. Less than must be transitive and non reflexive, but need not define a complete ordering. Less than returns true iff its first argument appears earlier in the (partial) ordering than the second argument, false if the second argument is earlier than the first and nil if the arguments are unordered. The built-in less than operation applies to enumeration types and to numbers.

Commentary

The built-in less than operator allows comparisons to be made between examples of ordered types such as enumeration types. For example:

```
# STATUS: current

olive: type is enum(large, jumbo, gigantic, colossal);
my_olive :: olive := colossal;
your_olive :: olive := jumbo;
confirm your_olive < my_olive;
```

Authors may declare new ordered types by defining a less-than operator. If the type is only partially ordered, it will be undefined for some values of the type, whereas if it is totally ordered, it will be defined for all values.

```
# STATUS: planned

type lieutenant;
type general;
type soldier is private ~ lieutenant ~ general;
type officer is lieutenant ~ general;
        "<"(x: officer, y: officer) is
                if x isa lieutenant & y isa general then
                        return true;
                else
                        return false;
        "<"(x: soldier, y: soldier) is
                if x isa private & y isa lieutenant | y isa general then
                        return true;
                else
                        return ?;
confirm any private < general & any private < lieutenant;
confirm any lieutenant < any general;
```

Here the type `officer` is fully ordered, because the **less-than** operator is defined for all values of `officers`, while the type `soldier` is partially ordered, since the **less-than** operators are only defined for `privates`.

The **less-than** operator must be transitive and non-reflexive, but need not define a complete ordering. It must return:

a. True if its first argument appears earlier in the (partial) ordering than the second argument
b. False if the second argument is earlier than the first
c. False if the two arguments appear at the same position in the ordering
d. ? if the arguments are unordered

4.1.3. ">"(x: ordered, y: ordered): boolean #{is y < x}#;

The **greater than** operation is automatically defined whenever less than is defined.

4.1.4. "<="(x: ordered, y: ordered): boolean #{is !(y < x)}#;

The **less than or equal** operation is automatically defined whenever less than is defined.

4.1.5. ">="(x: ordered, y: ordered): boolean #{is !(x < y)}#;

The **greater than or equal** operation is automatically defined whenever less than is defined.

4.2. Scalar Types

Any type whose examples are fully ordered is a scalar type. Scalar types include enumerations (5.7.2), machine integers (4.3.1), numbers (5.1.1), integers, and real numbers. Integers and real numbers are not supported. Complex numbers are not scalars.

4.2.1. scalar: ordered type;

A scalar type is an ordered type whose values form a fully ordered sequence.

Commentary

The soldiers example in 4.1.2 illustrates the difference between scalar and partially ordered types; here is another example:

```
# STATUS: planned

totally_ordered(i: int): scalar;
partially_ordered(i: int): scalar;
T3: totally_ordered := totally_ordered 3;
T9: totally_ordered := totally_ordered 9;

"<"(x: totally_ordered, y: totally_ordered): boolean is
        return (x.i < y.i);

"<"(x: partially_ordered, y: partially_ordered): boolean is
        if (x.i < 1000 & y.i < 1000) then
                return true;
        else
                return ?;

confirm T3 < T9;
```

Here the types `totally_ordered` and `partially_ordered` are identical except for the fact that the **less-than** operator for `partially_ordered` is undefined for values above 1000.

4.2.2. min(vector...): any;

The **min** operation applies to one or more arguments of the same type. It returns the minimum value among its arguments as defined by their ordering. If there are vector arguments, they must all be of the same dimension, and scalar arguments will be applied to all dimensions of the vectors.

Commentary

```
# STATUS: current

rainbow: type is enum(r, o, y, g, b, i, v);
smallest :: rainbow := min(y, b, o, i);
```

```
    biggest :: rainbow := max(y, b, o, i);
    confirm smallest = o;
    confirm biggest = i;
    confirm min(vector(0, 2, 13), 4, vector(7, 0.3, 10)) = vector(0, 0.3, 4);
```

4.2.3. max(vector...): any;

The **max** operation applies to one or more arguments of the same type. It returns the maximum value among its arguments as defined by their ordering. If there are vector arguments, they must all be of the same dimension, and scalar arguments will be applied to all dimensions of the vectors.

Commentary

```
    # STATUS: current

    confirm max(vector(0, 2, 3), 4.1, vector(7, 1, 0)) = vector(7, 4.1, 4.1);
```

4.3. Machine Integers

Machine integers are a built-in enumeration type for the integers in the range -32768 .. 32767. Machine integers all have literals as defined in A.3 . Relational operations (see 4.1), enumeration operations (see 5.7) and the operations of this section apply to machine integers. Machine integers may be used anywhere numbers are required. Numbers that are integers in the range of machine integers may be used anywhere machine integers are required. It is illegal to use any other number where an integer is required. The bit by bit operations of Section 4.4 operate on integers.

4.3.1. int: number type;

The definition of int in the current implementation is:

```
    int: enumeration type is
            # integer in -2^15.. 2^15-1
            property int << number;
```

Commentary

The type **int** represents machine integers. Machine integers may be used anywhere numbers are required, and numbers that are integers may be used anywhere machine integers are required. It is illegal to use any other number where an integer is required.

```
    # STATUS: planned

    a :: number := 1.234;
    b :: number := 1.0;
    i :: int := 3;
    j :: int := a; # Illegal
    k :: int := b; # Legal
    l :: int := 1.234; # Illegal
    c :: number := i; # Legal
```

4.3.2. indexer: enumeration int type;

The type indexer is the type of all integers in the range -2^15..2^15-1.

Commentary

The primary use of indexer is to declare indexing operations in the Easel Package Standard.

```
# STATUS: current

L: (list int) := new list int;
push(L, 1, 2, 3, 4, 5);
confirm L[4.0-1.0] = 4;
```

4.3.3. permute(n: indexer): list;

This routine returns a random permutation of the integers 0.. n-1. The resulting values may be used as indices on any list of `n` elements to randomly permute the order of reference.

Commentary

This routine returns a random permutation of the integers 0 through n-1. The resulting values may be used as indices on any list of `n` elements to permute the order of reference randomly.

```
# STATUS: current

l :: list := new list any;

l := permute 5;
confirm (length l) = 5;
for i: each (0..4) do
        confirm l[i] >= 0;
        confirm l[i] < 5;
```

In the following statements, the list shuffle is being used to scramble the hand upon output.

```
# STATUS: current

deal(): action is
        face_cards: type is enum(jack, queen, king, ace);
        hand: type is list face_cards;
        shuffle :: list := new list any;

        shuffle := permute 4;
        H :: hand := new hand;
        H := hand'[jack, queen, king, ace];
        output(H, "\cr");
        output(shuffle, "\cr");
        for i: each (0..3) do
                # prints e.g. king queen ace jack
                output(H[shuffle[i]], "\cr");
deal();
```

4.4. Bitwise Operations

Commentary

The following example illustrate the use of bit-wise operations. The first two comment lines define the equivalence between two decimal and binary numbers being used. The next line shows how pair-wise ANDing of the binary bit strings of these two numbers results on the binary number 1000 (8 decimal).

```
# STATUS: current

# 10 (decimal) = 1010 (binary)
# 12 (decimal) = 1100 (binary)
confirm band(10, 12) = 8;
confirm bor(10, 12) = 14;
confirm bxor(10, 12) = 6;
confirm (bnot 10) = (-11);
confirm bsr(10, 2) = 2;
confirm bsc(10, 12) = 40960;
confirm bsa(10, 2) = 2;
confirm bsa((-10), 2) = (-3);
```

4.4.1. band(int, int): int;

Perform a bit-wise and operation on the integers.

4.4.2. bor(int, int): int;

Perform a bit-wise or operation on the integers.

4.4.3. bxor(int, int): int;

Perform a bit-wise exclusive or operation on the integers.

4.4.4. bnot(int): int;

Complement each bit in the integer.

4.4.5. bsr(x: int, n: int): int;

Shift x right by n bits.

4.4.6. bsl(x: int, n: int): int;

Shift x left by n bits.

4.4.7. bsc(x: int, n: int): int;

Circular shift the first integer n bits to the right.

Commentary

Left circular shifts may be done by bsc 16-n. For example, a left shift 7 of 3 can be done by

```
bsc(3, 9);
```

4.4.8. bsa(x: int, n: int): int;

Perform a (non-circular) right shift, preserving the sign bit.

4.4.9. first_bit_different(int, int): int;

The first_bit_different operator compares the bits in the two integers and returns the bit position of the most significant bit that is different. The least significant bit is bit 0. If the two integers are identical, first_bit_different returns -1.

Commentary

Note that although the first bit is determined starting at the most significant bit, to accommodate different integer widths, the bit position returned is counted starting from the least significant bit.

Here is an example of first_bit_different:

```
# STATUS: current

confirm first_bit_different(01001#2, 01100#2) = 2;
```

The two integers differ in the third bit from the right, so **first_bit_different** returns 3.

Note that if one of the integers is 0, first_bit_different can be used as a first_bit_set function.

```
# STATUS: current

confirm first_bit_different(01000#2, 0) = 3;
```

Chapter 5. Numbers

Numbers are imprecise real values. Numbers are also fully ordered scalar values. Special lexical syntax is provided for numbers (A.3) and certain frequently constants are defined (5.1). Numeric operations include basic operations (5.2), integer conversion operation (5.3), trigonometric and hyperbolic functions (5.4), and statistical functions (5.5).

Vectors are sequences of numbers. Easel makes no distinction between length-1 vectors and numbers; thus, arithmetic aperations such as ***** and **+** are defined on vectors, but can be used on numbers.

5.1. Floating Point Numbers

5.1.1. number: scalar type;

Numbers are implemented in 32 bit IEEE floating point representation. This representation is precise to one part in 2^24 and has an epsilon of 1.19209209e-7, minimum positive value of 1.17549435e-38 and maximum positive value of 3.40282347e38. IEEE floating point also has special values for zero, infinity, and -infinity, as well as a variety of not_a_number values.

Commentary

Example: The following statement indicates that x can take on any value in the inclusive range -1.17549435e38 to 1.17549435e38.

```
# STATUS: current

x :: number := ?;
```

5.1.2. vector(number...): immutable type;

The type **vector** is the type of all n-dimensional vectors. A **number** in Easel is indistinguishable from a 1-dimensional vector.

5.1.3. infinity: number is 1.0/0.0;

Infinity is a pseudo number used in IEEE floating point to represent a value in excess of those otherwise representable. Computations may be done on infinity. Negative infinity as -infinity is also supported.

Commentary

```
# STATUS: current

a :: number := (-infinity) * (-infinity);
b :: number := infinity + (-infinity);
c :: number := 1.0/0.0;
d :: number := 0.0 - infinity;

confirm a = infinity;
confirm is_NaN b;
confirm c = infinity;
if (d = (-infinity)) then # Test for infinity
```

```
        output "ok";
```

The above examples illustrate how **infinity** is manipulated.

5.1.4. not_a_number: number is sqrt(-1);

Not_a_number is one of a set of pseudo numeric values used in IEEE floating point to indicate erroneous or non representable computational results. Many of the operations of this chapter can generate **Not_a_number** results and most will if given a **Not_a_number** argument.

Commentary

Not_a_number can be used to write numeric functions which return special values in error conditions. For example, the function `silly` below only operates on positive values.

```
# STATUS: current

x :: number := not_a_number;

silly(n: number): number is
        if n < 0.0 then
                return not_a_number;
        return n + 5.0;

output silly (-3.0); # prints "not_a_number"
```

5.1.5. is_NaN(number): boolean;

This operator returns true if its argument is not a number by IEEE rules.

Commentary

There are many **Not_a_number** values, and IEEE rules state that any equality test with a **Not_a_number** value must fail. In particular, **Not_a_number** is not equal to itself:

```
# STATUS: current

confirm not_a_number != not_a_number;
```

To allow for checking of **Not_a_number** values, the is_NaN function is provided.

```
# STATUS: current

f(n: number): string is
        if is_NaN n then
                return "NaN";
        else
                return "number";

confirm (f sqrt(-10.0)) == "NaN";

confirm is_NaN sqrt(-3.0);

confirm is_NaN 100.0 + sqrt(-3.0);

confirm is_NaN not_a_number;
```

The above example illustrates how is_NaN can be used.

5.1.6. e: number is 2.7182818;

For convenience, Easel predefines a few useful numeric constants. The base of the natural logarithms **e** is defined as 2.7182818.

5.1.7. pi: number is 3.14159265;

The value **pi** is defined as 3.14159265.

5.2. Basic Numeric Operations

Many of Easel's numeric operations are conventional, and are given without comment in this and the following sections.

Commentary

The use of these operations is shown in the following example.

```
# STATUS: current

numeric_example(): action is
        a :: number := 0.0006;
        b :: number := 0.2;
        c :: number := (-310.0);
        d :: int := 3;
        e :: number := 5.1234;
        x :: number := a+b/(c-d^e);
        confirm x = 2.600261e-4;
numeric_example();
```

5.2.1. itn(any type): any type;

If the parameter is an integer, **itn** returns the equivalent number. Otherwise, it does nothing.

Commentary

```
# STATUS: current

n :: number := itn 3;
confirm n = 3.0;
```

5.2.2. nti(any type): any type;

If the parameter is a number, **nti** returns the equivalent integer. Otherwise, it does nothing.

Commentary

In general, Easel users do not need to concern themselves with the internal representation of numeric quantities, as Easel performs conversions as necessary. The **nti** operator is for the rare cases where it is desirable to force a representation conversion.

```
# STATUS: current

n :: int := nti 3.0;
confirm n = 3;
```

5.2.3. "+"(vector, vector): vector;

5.2.4. "-"(vector, vector): vector;

5.2.5. "*"(vector, vector): vector;

5.2.6. "/"(vector, vector): vector;

5.2.7. "^"(vector, vector): vector;

5.2.8. "+"(vector): vector & it >= 0;

Returns the absolute value of the vector.

5.2.9. "-"(vector): vector;

5.2.10. "*"(vector): -1|0|1;

Returns the sign of the vector.

5.2.11. sqrt(vector): vector;

5.2.12. ln(vector): vector;

5.2.13. exp(vector): vector;

The exponential function.

5.2.14. log(x: vector): vector #{ is (ln x)/ln 10}# ;

Commentary

The following example illustrates the use of the functions described in this section.

```
# STATUS: current

numerics_example(): action is
        w :: number := sqrt 2500.0;
        x :: number := ln e;
        y :: number := log 100.0;

        confirm w = 50.0;
        confirm x = 1.0;
        confirm y = 2.0;
        output(x, "\cr", y, "\cr", w, "\cr");
numerics_example();
```

5.3. Integer Conversion Operations

Integer conversion operations return integral values, i.e. real numbers that have zero as the fractional value. Because the range of the numeric representation is imprecise, integral values are precise only in the range -32768.. 32767.

5.3.1. floor(vector): vector;

Floor returns the largest integer less than or equal to its argument.

Commentary

Examples of the use of floor are:

```
# STATUS: current

confirm (floor 123.45) = 123;
confirm (floor (-123.45)) = (-124);
```

5.3.2. ceil(vector): vector;

Ceil returns the smallest integer greater than or equal to its argument.

Commentary

Examples of the use of ceil are:

```
# STATUS: current

confirm (ceil 123.45) = 124;
confirm (ceil (-123.45)) = (-123);
```

5.3.3. round(x: vector, y: vector): vector;

Round returns the multiple of `y` which is nearest to `x`. Values exactly between multiples of `y` are rounded up when `y` is positive, and rounded down when `y` is negative. The formula for round is floor(x/y + 0.5) * y.

Commentary

There are four types of round, depending on whether values that are exactly between multiples are rounded up, down, away from zero, or towards zero. These four can all be written in Easel as follows, assuming that `B` is positive:

```
round(A, B);                 # rounds up
round(A, -B);                 # rounds down
round(+A, B) ** A;         # rounds away from 0
round(+A, -B) ** A;  # rounds towards 0
```

```
# STATUS: current

round_example(): action is
        A  :: number := 1.5;
        B :: number := 3.0;

        confirm round(A, B) = 3;                 # rounds up
        confirm round(A, 0-B) = 0;                 # rounds down

        A := 0.0 - A;

        confirm (round((+ A), B) * (* A)) = 0-3;         # rounds away from 0
        confirm (round((+ A), -B) * (* A)) = 0;          # rounds towards 0

round_example();
```

5.3.4. round(x: vector): indexer;

This form of round is the same as the first with `y = 1`, that is, it rounds towards **+infinity**. The formula for **round** is floor(x + 0.5).

Commentary

Examples of the use of round are:

```
# STATUS: current

confirm (round 123.45) = 123;
confirm (round (-123.45)) = -123;
```

In the first form, round returns the integer nearest to its argument with 0.5 values rounded down (i.e., round(x) = floor(x+0.5)). In the second form, round returns the multiple of y which is nearest to x with value exactly between multiples of y rounded down. The two forms are the same when y = 1.

5.3.5. trunc(x: vector): indexer; #{ is *x*floor (+ x) }#

Truncate returns the value of its argument without its fractional value. Truncate reduces positive values and increases negative values.

Commentary

Definition: trunc(x: vector): int is *x * floor (+ x)

The trunc operator returns the value of its argument without its fractional value, thus reducing positive values and increasing negative values.

Examples of the use of trunc are:

```
# STATUS: current

confirm (trunc 123.45)  = 123;
confirm (trunc (-123.45)) = (-123);
```

5.3.6. div(x: vector, y: vector): vector #{ is floor x/y }#;

Integer division returns the number of integral times y divides x.

Commentary

The following examples illustrate the use of integer division

```
# STATUS: current

confirm div (6.23,2.15) = 2.0;
confirm div (6.23, (-2.15)) = (-3.0);
```

5.3.7. mod(x: vector, y: vector): vector;

Mod returns the remainder from integer divide. The result is always non negative.

Commentary

The mod operator returns the remainder from integer divide. The result has the same sign as y.

The following illustrates the use of the mod function:

```
# STATUS: current

confirm mod(12, 5) = 2.0;
confirm mod(-12, 5) = 3.0;
confirm mod(12, (-5)) = -3.0;
confirm mod((-12), (-5)) = -2.0;
```

5.3.8. rem(x: vector, y: vector): vector;

Rem returns the remainder from integer divide. The result is negative when x and y have different signs.

Commentary

The following illustrate the use of the rem function

```
# STATUS: current

confirm rem(12, 5) = 2.0;
confirm rem(-12, 5) = -2.0;
confirm rem(12, (-5)) = 2.0;
confirm rem((-12), (-5)) = -2.0;
```

5.4. Trigonometric and Hyperbolic Functions

Trigonometric hyperbolic functions convert from angles to ratios of lengths which are unitless. Arc functions convert from ratios of lengths to angles. Although there exist implementations of trigonometric and hyperbolic functions without units, each function has many implementations depending on whether its parameter cycles at 360, 2 **pi**, or some other value. The functions described below will accept any unit of angle.

Commentary

The following examples illustrate the use of the trigonometric functions:

```
# STATUS: current

confirm (sin(pi / 6)) = 0.5;
```

The following examples illustrate the use of the hyperbolic functions:

```
# STATUS: current

confirm (sinh (-2.4)) = (-5.46623);
confirm (atanh 0.0) = 0.0;
confirm (atanh 0.5) = 0.549306;
```

5.4.1. sin(Angle): number;

5.4.2. cos(Angle): number;

5.4.3. tan(Angle): number;

5.4.4. asin(number): Angle;

Commentary

Note that -1.0 <= argument <= 1.0.

5.4.5. acos(number): Angle;

Commentary

Note that -1.0 <= argument <= 1.0.

5.4.6. atan(x: Length, y: Length): Angle;

Returns the arctangent of x/y.

Commentary

The one-parameter arctangent can be had by passing 1.0 as `y`. The two-parameter version is provided because it is often more convenient when dealing with coordinates.

```
# STATUS: current

confirm atan(1.0, 1.0) = pi/4.0;
confirm atan(infinity, 1.0) = pi/2.0;
confirm atan(infinity, 1.0) = atan(1.0, 0.0); # Equals pi/2
```

5.4.7. atan(v: vector): number;

Returns the arctangent of v[0]/v[1]. If the vector is one-dimensional, returns the artangent of v[0]/1.0.

Commentary

```
# STATUS: current

confirm (atan vector(1.0, 1.0)) = pi/4.0;
confirm (atan 1.0) = pi/4.0;
confirm (atan infinity) = pi/2.0;
confirm (atan infinity) = atan vector(1.0, 0.0); # Equals pi/2
```

5.4.8. sinh(number): number;

5.4.9. cosh(number): number;

5.4.10. tanh(number): number;

5.4.11. asinh(number): number;

5.4.12. acosh(number): number;

5.4.13. atanh(number): number;

5.5. Statistical Functions

Statistical calculations and simulations require random number generation with a variety of distributions. All of the random number functions of this section have the general form:

5.5.1. random(distributions, (any type)...): number;

where distribution is one of the random distributions of 5.5.2 and the remaining arguments conform to the requirements of the particular distribution as described in 5.5.3 through 5.5.16. Fourteen random distributions are provided. The seed for the statistical functions is a system value that may referenced and assigned by using 5.5.18 and 5.5.19.

Commentary

The distributions included are: beta, binomial, chi_square, exponential, f_distribution, gamma, lognormal, normal, poisson, triangular, uniform, weibull, continuous, and discrete. Of these, the distributions beta, binomial, chi_square, exponential, f_distribution, gamma, and poisson were derived from functions developed by the Department of Biomathematics, University of Texas.

Note that random distributions over enumeration types (including ints) also exist. These are discussed in 5.7.12.

The behavior of the random number functions of this section may be controlled by using the set_seed and get_seed procedures (see and).

5.5.2. distributions: enum(beta, binomial,
chi_square, exponential, f_distribution, gamma,
lognormal, normal, poisson, triangular,
uniform, weibull, continuous, discrete);

5.5.3. random(beta, a: number, b: number); number;

This function returns a random value from a beta distribution where the density of the beta function is

```
f(x) = x^(a-1) *
(1-x)*(b-1)/beta(a,b) for 0<x<1,
```

and where beta is the complete beta integral, random(chisq, df). The beta distribution takes many shapes (from linear to bell-shaped), depending on the values of `a` and `b`. This ability to take on many shapes makes the beta distribution useful if accurate empirical data are not available or if there is a need to define a probabilistic distribution that takes on a wide variety of characteristics.

5.5.4. random(binomial, n: indexer, p: number): number;

This routine returns the number of successes `k` in a series of trials `n` where the outcome of a successful trial is `p`. The result will be integral. For example, using this distribution, the function would return a number of heads (0, 1, 2, 3 or 4) in 4 tosses (where `p`=0.5) based on the computed probabilities of these occurrences (1/16, 4/16, 6/16, 4/16, and 1/16 respectively). This discrete distribution could be used to predict the number k of defective items in a production run of n units, given a probability p of one item being defective.

5.5.5. random(chi_square, df: number): number;

This function returns a random value from a chi square distribution where df is the degrees of freedom.

5.5.6. random(exponential, mean: number): number;

This function returns a random value from an exponential distribution with the given mean. The exponential distribution is predominantly used to define time intervals between random events, such as the time between customers arriving at a queue.

5.5.7. random(f_distribution, dfn: number, dfd: number): number;

This function returns a random value from an f (i.e., variance ratio) distribution where dfn is the degrees of freedom in the numerator and dfd is the degrees of freedom in the denominator.

5.5.8. random(gamma, a: number, r: number): number;

This function returns a random value from a gamma distribution whose density is

```
f(x) = ((a^-r)/gamma r) * x^(r-1) * e^(-a*x).
```

The gamma function is often used to represent the time to perform a task. Its shape can vary from exponential (r = 1) to a skewed bell-shaped curve (r = 2)

5.5.9. random(lognormal, scale: number, shape: number): number;

This function returns a random value from a lognormal distribution with the given scale and shape. If x has distribution random(lognormal, scale, shape), then ln x has distribution random(normal, scale, shape).The distribution is defined by the scale and shape parameters. This distribution is frequently used to define the duration of an activity, or the time between failures. Its shape is that of a skewed bell curve with the right side of the curve asymptotically converging to zero.

5.5.10. random(normal, mean: number, std_deviation: number): number;

This function returns a random value from a normal distribution with a given mean and standard distribution.This is the Gaussian or bell-shaped curve. It is often used to describe processes in which large numbers are involved and in which there is a symmetry about the mean. Its shape is defined by the mean and standard deviation.

5.5.11. random(poisson, mean: number): number;

This is a discrete distribution that can be used to compute the number of random events that occur is a fixed period. It can also be used to simulate batch sizes. Its shape is defined by the expected mean rate.

5.5.12. random(triangular, min: number, mode: number, max: number): number;

This function returns a random value of triangular distribution with given minimum, mode and maximum.The triangular distribution is used when a more accurate distribution is unknown, but where guesses at the minimum value (min), maximum value (max) and most probable value (mode) of the independent variable can be made.

5.5.13. random(uniform, low: number, high: number): number;

This function returns a random value from a uniform distribution between low and high. More precisely, the result may equal low but not high.

5.5.14. random(weibull, shape: number, scale: number): number;

This function returns a random value from a Weibull distribution with the given shape and scale.The Wiebull distribution can be used to represent to lifetime of a system. If we assume that the system's components fail independently, then the time between successive system failures can be approximated by the Weibull distribution. Like the gamma distribution the parameters shape and scale allow the shape of this distribution to vary widely.

5.5.15. random(continuous, cp: list number, x: list number): number;

This function returns a random value from a continuous distribution by linear interpolation from a user specified cumulative distribution. The list `x` specifies the `x` values for the cumulative distribution over the domain zero to one, where the last value must be one. The list cp specifies the corresponding cumulative probability values (where the last value must also be one). This continuous function can be used when a desired distribution does not match any of the analytic distributions.

5.5.16. random(discrete, cp: list number, x: list number): number;

This function returns a random value from a discrete distribution over a user specified cumulative distribution. The list `x` specifies the `x` values for the cumulative distribution over the domain zero to one, where the last value must be one. The list cp specifies the corresponding cumulative probabiluty values (where the last value must also be one). This discrete function can be used when a desired distribution does not match any of the analytic distributions.

5.5.17. random(dim: int): vector;

This function returns a vector whose dimensinality is **dim**, and whose elements are independent uniformly distributed random numbers in the range 0 .. 1.

Commentary

```
# STATUS: current

output 20 + 10 * random 5;
outln floor 22 * random 7;
```

The first example produces 5 random numbers between 20 and 30, while the second generates 7 random integers in the range 0 to 21.

5.5.18. get_seed() : (list int);

Returns the current value of the random number seed.

Commentary

The system generates a random seed automatically by calling `new_seed();` (see 5.5.20) when the application is initiated and may be different for each execution. The get_seed operation allows the program to save the current seed for later use to guarantee repeatable results.

5.5.19. set_seed(list) : action;

The **set_seed** operation sets the random number seed to the specified value.

Commentary

The **set_seed** operation allows the random seed to be set at any time under program control. The number may be a value returned by get_seed or computed by any other appropriate means.

In the following example, the random numbers generated in the two loops are the same since they are run using the same seed.

```
# STATUS: current

set_seed_example(): action is
        seed :: list := get_seed();
        L1 :: list := new list number;
        L2 :: list := new list number;
        set_seed(seed);
        for i: each (0..100) do
                push(L1, random(uniform, 0.0, 10.0));
        set_seed(seed);
        for i: each (0..100) do
                push(L2, random(uniform, 0.0, 10.0));
        confirm L1 == L2;
set_seed_example();
```

5.5.20. new_seed() : action;

The **new_seed** operation sets the random seed to a new, randomly-chosen value.

Commentary

In the following example, the random numbers generated in the two loops are different since they are run using different seeds.

```
# STATUS: current

set_seed_example(): action is
        L1 :: list := new list number;
        L2 :: list := new list number;
        new_seed();
        for i: each (0..100) do
                push(L1, random(uniform, 0.0, 10.0));
        new_seed();
        for i: each (0..100) do
                push(L2, random(uniform, 0.0, 10.0));
        confirm L1 != L2;
set_seed_example();
```

5.5.21. reset_seed() : action;

The **reset_seed** operation sets the random seed back to its initial value.

Commentary

In the following example, the random numbers generated in the two loops are the same since the seed is reset before each of them.

```
# STATUS: current

reset_seed_example(): action is
        L1 :: list := new list number;
        L2 :: list := new list number;
        reset_seed();
        for i: each (0..100) do
                push(L1, random(uniform, 0.0, 10.0));
        reset_seed();
        for i: each (0..100) do
                push(L2, random(uniform, 0.0, 10.0));
        confirm L1 != L2;
reset_seed_example();
```

5.6. Vector Operations

5.6.1. zero_vector(n: indexer): vector;

Returns a vector with dimensionality `n` whose values are all **0**.

Commentary

```
# STATUS: current

confirm (zero_vector 3) == vector(0.0, 0.0, 0.0);
```

5.6.2. unit_vector(n: indexer, i: indexer): vector;

Returns a vector with dimensionality `n` whose values are all **0** except for the **i**th value, which is 1.0.

Commentary

```
# STATUS: current

confirm unit_vector (3, 1) == vector(0.0, 1.0, 0.0);
```

5.6.3. block_vector(n: indexer, r: number): vector;

Returns a vector of dimensionality **n** whose values are all **r**.

Commentary

```
# STATUS: current

confirm block_vector(3, 9) = vector(9, 9, 9);
```

5.6.4. dim(vector): indexer;

Returns the dimensionality of the specified vector.

Commentary

```
# STATUS: current
confirm (dim vector(6, 9, 3, 4, 1)) = 5;
```

5.6.5. len(vector): number;

Returns the Cartesian length of the vector, i.e. the distance from the origin to the point specified by the vector.

Commentary

A vector in Easel is simultaneously a point, a direction (from the origin to the point), and a distance (from the origin to the point).

```
# STATUS: current
confirm (len vector(4, 3)) = 5;
```

5.6.6. sqrlen(vector): number;

Returns the square of the Cartesian length of the vector.

Commentary

```
# STATUS: current
confirm (sqrlen vector(4, 3)) = 25;
```

5.6.7. normalize(vector): vector;

Returns the vector divided by its Cartesion length; the length of the result is 1.0;

Commentary

```
# STATUS: current
confirm (normalize vector(4, 3)) = vector(4/5, 3/5);
```

5.6.8. dot(vector, vector): number;

Returns the dot product of the two vectors.

Commentary

```
# STATUS: current
outln dot(vector(3, 4), vector(4, 5));
```

5.6.9. cross(vector...): vector;

Returns the cross product of the specified vectors

Commentary

```
# STATUS: current

outln cross(vector(3, 4), vector(4, 5));
```

5.7. Enumeration Types

An enumeration type is a scalar type with a finite number of examples. The enumeration type constructor operation (see 5.7.2) is used to define an enumeration type by listing the names of its examples in order. Machine integers (see 4.3.1) are examples of enumeration types.

5.7.1. enumeration: scalar type;

The type **enumeration** is the supertype of every enumeration type.

5.7.2. enum(identifier...): scalar type;

The **enum** operation constructs a scalar type by enumerating the names of its examples in order. Scalar types are described in 4.2. A side effect of a call on enum is to define those identifiers with their corresponding values throughout the scope of the current simulation.

Enumerations are treated as cycles, with first immediately following last. This affects the successor, predecessor, and enum ranges operations, but does not affect the relational operations.

Commentary

```
# STATUS: current

primary_color: type is enum(r, y, b);
```

Here the enumeration type `primary_color` is being constructed by listing the names `r`, `y`, and `b` in order.

Calls on the enumeration construction operation define the given identifiers with their corresponding values throughout the scope of the current simulation. For example, in the following statements the identifier `Vanilla` is defined by the call on enum, and is then referenced in the declaration of Sundae:

```
# STATUS: current
```

Since enumerations are treated as a cycle the successor of the last element is the first element. This is useful when dealing with repeating sequences of values. For example, the days of the week constitute a repeating sequence, since the successor of Sunday is Monday:

```
# STATUS: current

day: type is enum(mon, tue, wed, thu, fri, sat, sun);
flavor: type is enum(Vanilla, Strawberry, Chocolate);
Sundae :: flavor := Vanilla;
confirm Chocolate + 1 = Vanilla;
```

```
    confirm sun + 1 = mon;
```

Here the successor of `Chocolate` is `Vanilla`, and the successor of `sun` is `mon`.

5.7.3. first(enumeration type): the type;

First returns the least value of the enumeration type.

Commentary

First and last return the first and last values of a specified enumeration type.

```
    # STATUS: current

    day: type is enum(mon, tue, wed, thu, fri, sat, sun);
    flavor: type is enum(Vanilla, Strawberry, Chocolate);
    confirm (first flavor) = Vanilla;
    confirm (last day) = sun;
```

Here first and last are being applied to the enumeration types flavor and day, respectively.

5.7.4. last(enumeration type): the type;

Last returns the greatest value of the enumeration type.

5.7.5. ord(enumeration): indexer;

Ord returns the ordinal value of the enumeration element.

Commentary

```
    # STATUS: current

    day: type is enum(mon, tue, wed, thu, fri, sat, sun);
    confirm (ord tue) = 1;
    confirm (ord sat) = 5;
```

5.7.6. min(enumeration...): any;

5.7.7. max(enumeration...): any;

5.7.8. "+"(x: enumeration, int): type_of x;

This operation returns the nth successor of its first argument modulo the number of elements in the enumeration. The indexer argument may be negative.

Commentary

```
    # STATUS: current

    day: type is enum(mon, tue, wed, thu, fri, sat, sun);
    flavor: type is enum(Vanilla, Strawberry, Chocolate, Butter_pecan, Mint);

    f1 :: flavor := Vanilla + 1;
    f2 :: flavor := Butter_pecan - 3;

    confirm f1 = Strawberry;
    confirm f2 = Vanilla;
    confirm mon + 8 = tue;
    confirm Vanilla + 8 = Butter_pecan;
```

Here the successor and predecessors are being computed for various flavors and days of the week.

This shows that the eighth day after `Monday` is `Tuesday`, and that the eighth flavor after `Vanilla` is `butter pecan`.

Of course, the **indexer** arguments may be negative:

```
    # STATUS: current

    day: type is enum(mon, tue, wed, thu, fri, sat, sun);
    flavor: type is enum(Vanilla, Strawberry, Chocolate);
    confirm mon + -2 = sat;
    confirm Chocolate + -2 = Vanilla;
```

These are equivalent to `mon - 2` and `Chocolate - 2`.

5.7.9. "-"(x: enumeration, int): type_of x;

This operation returns the nth predecessor of its first argument modulo the number of elements in the enumeration. The indexer argument may be negative.

5.7.10. "-"(x: enumeration, type_of x): indexer;

This operation returns the number of elements from its first to second arguments in the order of the enumeration.

Commentary

```
    # STATUS: current

    olive: type is enum(small, medium, large, giant, jumbo, collossal);
    season: type is enum(spring, summer, autumn, winter);
    confirm jumbo - medium = 3;
    confirm small - jumbo = -4;
    confirm winter - spring = 3;
```

In this example `jumbo` comes 3 positions after `medium`, while `small` comes 4 positions before `jumbo`.

5.7.11. ".."(x: enumeration | int, type_of x): (type_of x) type;

The operation returns a range of an enumeration type.

Commentary

```
    # STATUS: current

    olive: type is enum(small, medium, large, giant, jumbo, collossal);
    season: type is enum(spring, summer, autumn, winter);
    outln medium .. jumbo;
    outln spring .. autumn;
    outln autumn .. spring;
    outln giant .. small;
```

Restrictions

Enumeration ranges are only implemented within for loops.

5.7.12. rand(lower: enumeration, upper: type_of lower): type_of lower;

This routine returns a random value from a range of an enumeration type. Recall however that int is an enumeration type (see 5.7).

Commentary

This form of random returns a value in the range lower..upper.

```
# STATUS: current

olive: type is enum(small, medium, large, giant, jumbo, colossal);
season: type is enum(spring, summer, autumn, winter);

output rand (small, colossal); # prints e.g. giant
output "\cr";
output rand (spring, autumn); # prints e.g. spring;
```

Here the function is being used to compute random `olives` in the range `small..colossal` and `seasons` in the range `spring..autumn`.

Since **int** is an enumeration type, the random enumeration function can be used to generate random integers.

```
# STATUS: current

output rand(0, 10000); # prints e.g. 5976
```

Here 5976 is returned as a random int in the range 0 and 10000.

The following example illustrates enumeration types. Note that if one enumeration subrange is contained in another, it is a subtype of the larger range.

```
# STATUS: current

main(): action is
# Define the rainbow colors
        rainbow: type is enum(r, o, y, g, b, i, v);
        my_colors: type is y..i;
        C :: rainbow := rand(r, v); # returns a random rainbow color
        confirm (first rainbow) = r;
        confirm (last rainbow) = v;
        confirm r + 3 = g;
        confirm g - 1= y;
        confirm b - r = 4;

main();
```

Chapter 6. Attributes

6.1. The Attribute Type

6.2. Attribute Definitions

6.2.1. control "_attr_def"(identifier, type, the type): property;

An attribute definition specifies the name of the attribute, the type over which its value may range and the initial value of the attribute when constructing an example. The same attribute may be multiply defined either within a given define body or through inheritance of properties from supertypes. In such cases, the type of the attribute is the specialization (i.e. union of properties) of the multiple specifications. Two attributes of the same name may not be defined in the same local scope.

Commentary

```
# STATUS: current

t1: type;
t2 :: type := int;
```

Here we declare a **type** `t1` and an attribute of type **type**. In this example, `t1` is a type with unknown properties, while `t2` is an attribute of type **type** whose initial value is int.

```
# STATUS: current
leaf_shape: type is enum(pinnate, palmate, lobed);
L :: leaf_shape := palmate;
```

In this example, the name of the attribute is `L`, the type over which its value ranges is `leaf_shape`, and the initial value is `palmate`.

Attributes may be acquired either through local definitions, or through inheritance.

```
# STATUS: current

polygon: type is
        sides :: int := 3;
colored_polygon: type is
        property polygon;
        C :: pattern := azure;
CP :: colored_polygon := new colored_polygon;

confirm CP.C = azure;
confirm CP.sides = 3;
```

Here colored_polygon acquires the attribute `sides` through inheritance, and the attribute `C` by means of a local definition.

The same attribute may be multiply defined either through inheritance of properties from supertypes, or by a local definition with the same name as an inherited one.

```
# STATUS: current

positive: type is
        int & > 0;
wheeled: type is
        num_wheels :: positive := ?;
cycle: type is
        property wheeled;
        num_wheels: 1 .. 4 := ?;
bicycle: type is
        property cycle;
        num_wheels: 0 .. 4 := 2;
training_bicycle: type is
        property bicycle;
        num_wheels :: positive := 4;
Bills_bike :: bicycle := new bicycle;
confirm Bills_bike.num_wheels = 2;
Bills_bike.num_wheels := 3;
# Bills_bike.num_wheels := 7; would be illegal
```

Here we show the inheritance relationships among various kinds of `wheeled vehicles`. `Wheeled vehicles` can have any number of `wheels` as long they have at least one, while `cycles` cannot have more than 4 `wheels`. A `bicycle` typically has two `wheels`, so that is its initial value for `num_wheels`, but may also have 3 or four wheels in exceptional cases.

In cases of multiple specifications for the same attribute, the type of the attribute is the specialization (i.e. union of properties) of the multiple specifications. In the example above, the attribute `num_wheels` in `Bill's bike` is equivalent to:

```
# STATUS: planned

Bills_bike: positive & 1..4 & 0..4 & positive;
```

which of course reduces to simply 1..4.

It is important to distinguish multiple specifications of the *same* attribute from the case where multiple *different* attributes of the same name are inherited. Consider the following example:

```
# STATUS: planned

military: type is
        C :: country := USA;
submarine: type is
        C :: color := yellow;
military_submarine: type is
        property submarine & military;
        C :: integer := 0;
MS :: military_submarine := new military_submarine;
```

Here the type `military_submarine` inherits two different attributes both named `C` from its supertypes, and declares a local one of its own. Easel will treat these as the same attribute, with the unintended result that C has no properties, since `country` and `color` and `integer` have no properties in common. Such name collisions must be avoided by the user.

Two attributes of the same name may not be defined in the same local scope.

```
# STATUS: current
```

```
bird: type;
noisy: type;
Tweety :: bird := ?;
i :: int := 3;
Tweety :: noisy := ?;
```

This is an error, because there are two attribute definitions in the same scope with the same name (Tweety).

If the name of an attribute is unspecified (see 2.6.1), the attribute is anonymous and can be referenced only by dot quantification.

```
# STATUS: current

test(): type is
        ? :: string := "abc";

T :: test := new test();
confirm T.1 == "abc";
```

In this example, the anonymous **string attribute** is referenced by quantification within the body of test, and by dot quantification on `T` at the outer level.

The most common use of anonymous attributes is in formal parameter attributes. In this case, the unspecified name can be omitted.

```
# STATUS: planned

publish(book): action is
        proof_read the book;
        print the book;
        publish the book;
```

This is equivalent to

```
# STATUS: planned

publish(?:book): action is
        proof_read the book;
        print the book;
        publish the book;
```

6.3. Attribute Initialization

Each attribute definition may specify an initial value for the attribute. The value is specified as an expression that is interpreted once per instantiation of the attribute's most local enclosing body. The unspecified indicator **?** (see 2.6.1) may be used as the initial value in any attribute definition to mean that the initial value is not being specified at that point.

The initial value specified in any local attr_def overrides any inherited initial value inherited from a supertype for the same attribute. If the initial value is unspecified locally, the specified values from all supertype specifications of the attribute will be used. If their are multiple initial value specification either from multiple local specification or when there is no local specification from multiple supertype specifications, all specification must have the same value or an error will be reported.

If in the body of an operation, a reference to an attribute appears before the attribute is initialized or assigned, the reference will generate an uninitialized attribute error. In the case of initialization the error will be at compile time, and for assignment generally at run time.

Commentary

Each attribute definition may specify an initial value for the attribute. Not providing the initial value is equivalent to providing the unspecified value "?". Thus

```
# STATUS: current

i: int;
```

is equivalent to

```
# STATUS: current

i :: int := ?;
```

Both these examples declare `i` to be an integer attribute with an unspecified initial value.

It is important to distinguish between attribute definitions without initial values, and type definitions without initial values. Attribute definitions must give a type, while type definitions must give a metatype (which, given the Easel restriction that prohibits named metatypes, must end with the keyword **type**). Consider:

```
# STATUS: current

i1: int;
i2: int type;
```

This is equivalent to the following:

```
# STATUS: current

i1 :: int := ?;
i2: int type is ?;
```

In other words, it is the presence of the word **type** that indicates that `i2` is a type definition with an unspecified value, and `i1` is an integer with an unspecified initial value.

The initial value of an attribute is specified as an expression that is interpreted once per instantiation of the attribute's most local enclosing body. The following example illustrates the point. (see 8 for discussion of simulations and actors):

```
# STATUS: current

counter: simulation type is
        count :: int := 0;

a(id: int): actor type is

        # Initialization - each actor executes once
        j :: int := ?;
        k :: int := sim.count;

        # Begin body
        j := sim.count;
        sim.count := sim.count + 5;
        output(id, " ", k, " ", j, "\cr");
        wait 1.0;
```

```
main(): action is
        s :: counter := new counter;
        for i: each (1..3) do
                null new(s, a(i));
        wait s;
main();

# Prints:
# 1 0 0
# 2 0 5
# 3 0 10
```

Here the main procedure creates three actors, each of which interprets the expression which initializes the attribute j to `sim.count` and then increments the counter.

The unspecified indicator **?** (see 2.6.1) may be used as the initial value in any attribute definition to mean that the initial value is not being specified at that point. This is illustrated in the example below:

```
# STATUS: current

S :: string := ?;
output S; # Exception - the string is unspecified
```

The initial value specified in any local attribute definition overrides any inherited initial value inherited from a supertype for the same attribute. For example:

```
# STATUS: current

shape: mutable type is
        sides :: int := 3;
colored: mutable type is
        hue :: pattern := red;
green_square: mutable type is
        property shape & colored;
        sides :: int := 4;
        hue :: pattern := green;
GS :: green_square := new green_square;
confirm GS.sides = 4;
confirm GS.hue = green;
```

If the initial value is unspecified locally, the specified values from all supertype specifications of the attribute will be used. For example:

```
# STATUS: current

shape: type is
        sides :: int := 3;
colored: type is
        hue :: pattern := red;
red_triangle: type is
        property shape & colored;
RT :: red_triangle := new red_triangle;
confirm RT.sides = 3;
confirm RT.hue = red;
```

If there are multiple initial value specifications, either from multiple local specifications or when there is no local specification from multiple supertype specifications, all specifications must have the same value or an error will be reported.

If no initial value is specified for an attribute, it is illegal to reference that attribute before it is assigned. For example:

```
# STATUS: current

p(): action is
        k :: int := m + 2; # Legal - m has an initial value
        j :: int := i + 1; # Illegal - i has no initial value
        i :: int := ?;
        m :: int := 5;
p();
```

If in the body of an operation, a reference to an attribute appears before the attribute is initialized or assigned, the reference will generate an uninitialized attribute error. In the case of initialization the error will be at compile time, and for assignment generally at run time.

```
# STATUS: current

p(): action is
        i :: int := 3;
        j :: int := ?;
        k :: int := ?;
        j := 5;
        outln i; # Legal - has initialization
        outln j; # Legal - assigned
        outln k; # Illegal - neither initialized nor assigned
```

For initialization of actor attributes, see 8.1.2.

Restrictions

Initial values not overridden correctly - an error is reported unless all values are the same. For example, initial_value reports 'Inconsistent multiple attribute `sides`'.

6.4. Formal Parameter Attributes

A formal parameter is a special form of attribute that is specified as part of the lambda definition (see 2.5.1) and initialized by application of the lambda (see 3.9.1 and). Formal parameters are attributes of both the lambda subtype and of all its examples. Formal parameters are attributes whose initial value is specified in a lambda application. They are non private but assignable from outside the body of the define.

Commentary

```
# STATUS: current

f(i: int): int is
        return i;

poly(sides: int): mutable type is
        surface :: number := 1.0;

main(): action is
        hexagon: type is poly 6;
        H :: poly := new hexagon;

        confirm (f 3) = 3;
        confirm H.sides = 6;

main();
```

Here `i` and sides are formal parameter attributes. They are initialized to 3 and 6 by the calls shown.

Formal parameters are attributes of both the lambda subtype and of all its examples.

```
# STATUS: planned

player(i: int): actor type is
        i := 5;
P :: player := new player;
confirm player has i;
confirm P has i;
```

Here `i` is an attribute of both the type `player` and of all examples of player.

Formal parameters are not private, but they are never assignable from outside the body of the define.

```
# STATUS: current

t1(i: int): type is
        j :: int := i;
g :: t1 := new t1 3;
g.i := 10; # Illegal!
```

6.5. Scope of Visibility for Attributes

An attribute is visible only within the most local enclosing scope of its definition. That scope may be either a body scope or a compound statement. Formal parameters are exceptional in that they are visible not only within the formal parameter list but in the body of the lambda definition.

Commentary

An attribute is visible only within the most local enclosing scope of its definition. That scope may be either a body scope or a compound statement. This implies that no function or procedure may directly reference attributes in other functions or procedures. It also implies that the lexical nesting of functions and procedures has no impact on their attribute visibility.

Here is an example that illustrates the visibility of attributes:

```
# STATUS: current

r(): action is
        k :: int := 2;
p(): action is
        i :: int := 3; # Visible only in p
        j :: int := 9;
        q(): action is
                output i;    # Illegal - i directly visible only within p
                output p.i;    # Legal - returns "int"
                output r.k;    # Legal - returns "int"
                output q.i;    # Legal - returns "int"
                output self.i; # Illegal unless your actor has an i
        for l1: each (1..5) do
                i :: int := 6;
                output i;      # OK - returns 6
                output j;      # OK - returns 9
                output this.i; # OK - returns 3
```

```
            output l1; # Illegal - l1 only visible within loop
            output i2; # Illegal - i2 only visible within loop
    p();
    output i;   # Illegal - i directly visible only within p
    output p.i; # Legal - returns "int"
```

Here `p's i` is not directly visible within the body of `q` even though `q` is nested within `p`, while the re-definition of `i` in the for loop hides the outer definition. Note that dot qualification can be used to access these attributes even when they are not directly visible, but it returns the types of the attributes, not their values.

Formal parameters are exceptional in that they are visible not only within the formal parameter list but in the body of the lambda function's definition as well. For example:

```
    # STATUS: current

    find_max(N: int, A: list int): int is
            max :: int := A[0];
            for I: each (1 .. N) do
                    if (A[I] > max) then
                            max := A[I];
            return max;
    confirm find_max(5, (list int)'[3, 27, (-2), 5, 0, 6]) = 27;
```

Note that `N` is used both in the formal parameter list of `find_max (to declare A)` and in the body of `find_max`.

6.6. Operations on Attributes

6.6.1. "ref"(cee any): any type;

The attribute reference operation returns the current value of the named attribute. The argument must be an identifier naming an attribute that is visible at the point of the reference. If the attribute has not been initialized, an error will be reported and the nil value returned. The visibility rules guarantee that attributes and defines of the same name are never simultaneously visible. Syntactically, the appearance of the identifier name alone constitutes a reference.

Commentary

The appearance of an attribute name outside its definition constitutes a reference to that attribute. The reference returns the current value of the named attribute. If the attribute has not been initialized, an error will be reported and the nil value returned.

```
    # STATUS: planned

    i :: int := 3;
    j :: int := ?;

    # 3 because initialized
    confirm i = 3;

    # nil because j is unspecified (uninitialized)
    confirm j = nil;
    # Currently j returns "?", not nil - one can write
    # "confirm j = ?", for example.
```

Restrictions

Uninitialized attributes return **?**, not **nil**.

6.6.2. control ":="(any, (any type)): action;

The **attribute assignment** operation replaces the value of an attribute by the value of its second parameter. The attribute must be variable, must have an identifier name and must be visible at the point of the assignment.

Commentary

The **attribute assignment** operation replaces the value of an attribute by the value of its second parameter. The attribute must be variable, must have an identifier name, and must be visible at the point of the assignment.

```
# STATUS: current

i :: int := 3;                                  # Attribute initialization
i := 6;                                       # Attribute assignment
vect: type is list int;
V :: vect := new vect;
V := vect'[56, 47, 23, 12, 98];             # Attribute assignment
confirm V[1] = 47;
```

6.6.3. control type_of(any): type;

The argument to type_of must be a reference to an attribute or an element of an enumeration type. It returns the attribute's type as specified in the attr_def or the enumeration type as specified by an enum constructor.

Commentary

Example:

```
# STATUS: planned

main(): action is
        size: type is enum(small, medium, large, extra_large);

        i :: int := 0;

        X :: type := type_of i;       # X is int
        Y :: X := 3;
        Z: (type_of i) := 5;        # Y and Z are examples of int
        S: (type_of medium) := small; # S is of type size

        confirm X << int;
        confirm Y isa int;
        confirm Z isa int;
        confirm S isa size;

main();
```

Note that, except in the case of enumeration types, when applied to a formal parameter type_of returns the type of the formal, not the type of the actual. Consider the following:

```
# STATUS: planned

main(): action is
        even: type is int & mod 2 = 0;

        p(a: int): action is
```

```
                x :: type := type_of a;
                confirm ! (x << even);

        twosie :: even := 4;
        threesie :: int := 3;

        p twosie;
        p threesie;
main();
```

Here both calls on confirm succeed, since x is `int`, not `even`, even when an even number is passed as the actual parameter. This is because in general examples do not belong to a single type.

The exception is that for enumeration types, type_of *does* return the type of the actual. Consider:

```
# STATUS: planned

main(): action is
        insect: type is enum(mite, flea, fly, beetle, moth);
        writing_instrument: type is enum(pencil, pen, stylus, keyboard);

        p(q: enumeration): action is
                x :: type := type_of a;
                if (x << insect) then
                        output "insect";
                else
                        output "non-insect";

        Bic :: writing_instrument := pen;
        Bea :: insect := beetle;

        p Bic;  # non-insect
        p Bea; # insect
main();
```

Here the type of `q` is not **enumeration**, but the type of the actual - `writing_instrument` for `Bic` and `insect` for `Bea`.

Restrictions

Type_of is used internally by the translator, but is not available at the user level.

6.6.4. "init_val"((any type)...): any type;

This is a special internal operation that is called implicitly when there are multiple potentially conflicting specifications of the initial value of an attribute, and the translator is unable to prove that they will produce the same value. Each value is passed as an argument to attr_init and the attribute initialized the return value of attr_init. Attr_init compares its arguments for equality. If they are all the same it returns one of them. Otherwise, it reports an error and returns nil.

6.7. Composite Types

6.7.1. composite: type;

A composite type is any type that has attributes, formal parameters, indexed attributes and external attributes. All records, lists and parameterized types are composite.

6.7.2. immutable: type;

An immutable type is any type whose examples do not have any attributes whose values change over time. This implies that attributes of immutables can be specified only as parameters to the type.

Commentary

Unless a type is explicitly declared immutable, the Easel system assumes it is mutable. Eventually the system will be able to detect discrepancies between the definitions and their bodies (e.g. a type that is defined as immutable but which has assignable attributes), but it does not do so currently.

In the following example, joy and unicorns are declared as immutable types, while monkeys are mutable, since they have tails whose length can vary

```
# STATUS: current

joy: immutable type;
unicorn: immutable type is
        horns: int is 1;
monkey: type is
        tail_length :: int := 40;

confirm joy << immutable;
confirm unicorn << immutable;
confirm (! (monkey << immutable));
```

Note that **is** assigns a permanent value to an attribute, while **:=** assigns a value that can change over time (unless the type of the attribute is a singleton type). Thus `k :: 3 := 3` and `k: int is 3;` are equivalent.

Other examples of immutables are integers, characters, other enumerations, and types.

6.7.3. "[ ]"((any type)...): composite;

The aggregate constructor returns a new composite value with one attribute for each actual parameter. Each attribute is initialized to the corresponding actual parameter value. Aggregates may be used to construct mutable or immutable values.

The potential syntactic ambiguity between indexing and application to aggregates is always resolved in favor of indexing. If application to an aggregate is desired, the aggregate must be parenthesized.

Commentary

```
# STATUS: current

# Types
student(id: string): type is
        name :: string := ?;
        GPA :: number := ?;
trio: type is list number;

fruit: type is enum(apple, cherry, banana, pineapple, mango, guava);

# Examples
S :: student := new student "56789";
T :: trio := new trio;
fruitopia :: list := new list any;

# Assignments
S := student'["12345", "John Greene", 3.4];
```

```
    T := trio'[3.4, 5.6, 7.8];
    fruitopia := list'[mango, pineapple, cherry];
    push(fruitopia, apple);

    # Confirmations
    confirm S.id == "12345";
    confirm S.name == "John Greene";
    confirm S.GPA = 3.4;

    confirm T[0] = 3.4;
    confirm T[2] = 7.8;

    confirm T == trio'[3.4, 5.6, 7.8];
    confirm S == student'["12345", "John Greene", 3.4];

    confirm fruitopia == list'[mango, pineapple, cherry, apple];
```

Restrictions

Assignment of aggregates to `records` requires a type qualification operator.

6.7.4. "."(composite, enumeration): any type;

Dot qualification returns the value of an attribute (including formal parameters) given a composite value and an identifier or enum literal. The second parameter must be the name of an attribute defined for the composite's type.

Dot qualification can also be used to reference list elements if the second parameter is a numeric literal; in this usage it is equivalent to an index operation.

Finally, dot qualification can also be used to reference the value of a parameter of a subtype of a parameterized type. If the composite does not have a local attribute of the given name but instead has a composite attribute with an attribute of the given name, the value of the indirect attribute will be returned. This latter process may be applied recursively.

Commentary

The second parameter must be the name of an attribute, definition, or formal parameter defined in the body of the composite's type.

```
    # STATUS: current

    main(): action is

            genre: type is enum(classical, rock, folk, country);
            medium: type is enum(lp, cd, cassette, dvd, mp3);

            musical_work(G: genre): type is
                    sampling :: int := 22000;

            hi_fi(mw: musical_work): boolean is
                    return mw.sampling > 20000;

            Yellow_Submarine :: musical_work := new musical_work rock;

            confirm Yellow_Submarine.sampling = 22000;
            confirm Yellow_Submarine.G = rock;
            confirm hi_fi Yellow_Submarine;

    main();
```

Here `M.sampling` is a dot qualified reference to an attribute, while `M.genre` is a dot qualified reference to a formal parameter and `M.hi_fi` is a dot qualified reference to the definition `hi_fi`.

```
# STATUS: planned

music(G: genre): actor type is
        genre: type is enum(classical, rock, folk, country);
        medium: type is enum(lp, cd, cassette, dvd, mp3);
        sampling: int := 22000 Hz;
        cd..dvd: int := 100;
        DVDs(): int is
                return self.dvd;
M is music rock;
confirm M.sampling = 22000; # Attribute
confirm M.G = rock; # Formal parameter
confirm M.DVDs() = 100; # Definition
```

Here is an example of using a numeric literal to dot qualify a list:

```
# STATUS: current

l: list := list'[98, 99, 100];

confirm l.2 = 100;
```

Dot qualification can also be used to reference the value of a parameter of a subtype of a parameterized type. If the composite does not have a local attribute of the given name but instead has a composite attribute with an attribute of the given name, the value of the indirect attribute will be returned. This latter process may be applied recursively.

```
# STATUS: planned

account: type is
        id :: string := "";
        balance :: number := 0.0;
student: type is
        A :: account := new account;
        classes :: list := new list any;

S :: student := new student;

confirm S.balance = 0.0;
confirm S.A.balance = S.balance;
```

Here the student `S` can be dot qualified to obtain his or her balance, even though the balance is "nested" inside the attribute `A`.

6.7.5. "index"(composite, indexer): any type;

The **indexed reference** operation references the value of an attribute given a composite having that attribute and an enumeration value specifying which attribute in intended. The potential syntactic ambiguity between indexing and application to aggregates is resolved in favor of indexing. If application to an aggregate is desired, the aggregate must be parenthesized.

Commentary

```
# STATUS: current

L: list := new list;
L := list'["my", "dog", "has", "fleas"];
confirm L[1] == "dog";
```

6.8. Mutable Types

6.8.1. mutable: composite type;

A mutable type is any type whose examples have an identity separate and apart from the values of their attributes. Mutable types may be distinguished by having state, by having assignable attributes, and by the need for an explicit operation, new, to create a reference to an example.

Commentary

```
# STATUS: current

school(string): type is
        classrooms :: int := 0;
S :: school := new school "Allderdice";
T :: school := new school "Allderdice";
U: school is T;

confirm S != T;
confirm U = T;
```

`S` and `T` in this example are unequal because they have an identity separate from their attributes, which are equal (`Allderdice` and `0`). `U` and `T` are equal because `U` is just a constant whose value is `T`.

6.8.2. "."(mutable, enumeration, any): action;

Dot qualified assignment is analogous to dot qualified reference except that the first argument must be a mutable and the value of the attribute is replaced rather than referenced. The third argument specifies the new value.

Commentary

Dot qualified assignment replaces the value of an attribute given a mutable example, the attribute name, and the new value. In the following, `S` is the example, `classrooms` is the name, and `108` is the new example:

```
# STATUS: current

school(name: string): type is
        classrooms :: int := 40;
S :: school := new school "Schenley";
S.classrooms := 108;

confirm S.classrooms = 108;
confirm S.name = "Schenley";
```

Restrictions

Currently, no type-matching is done between the formal and actual parameters to the assignment operator.

6.8.3. new(mutable type): the type;

New creates a reference to an example of a mutable type. Each new reference is unique and distinguishable from all other references. Multiple references however may refer to the same example.

Commentary

```
# STATUS: current

school(string): mutable type is
        classrooms :: int := 0;
S :: school := new school "Allderdice";
T :: school := new school "Allderdice";
confirm S != T;
```

Here `S` and `T` are two unique, distinguishable references to the same example.

```
# STATUS: current

node(ID: int): type is
        links: list int := new list int;
network: type is list node;
Net :: network := new network;
for i: every (0..9) do
        push(Net, new node i);
        for j: each 0..9 do
                push(Net[i].links, rand(0, 99));
for i: each (0 .. last Net) do
        for j: each (0 .. last Net[i].links) do
                outln(i, " ", j, ": ", Net[i].links[j]);
```

In the above example a `network` of 100 new nodes is created. Each node is linked randomly to 10 other nodes in the `network`.

```
# STATUS: current

network: simulation type is
        initialized: boolean := false;

routingTable: type is list list int;

router(id: int): actor type is
        RT :: routingTable := new routingTable;
        when sim.initialized;
        confirm RT[0] == list'[100];
        confirm RT[3] == list'[103];

initiateRoutingTable(r: router): action is
        for i: each 0..14 do
                repln("r.RT is ", r.RT);
                push(r.RT, new list int);

        for i: every 0..14 do
                push(r.RT[i], 100+i);

n: network := new network;
r :: router := new(n, router 99);
initiateRoutingTable r;
n.initialized := true;
wait n;
```

In this example, a list of lists is initialized by calling a procedure to allocate each list and push a value on it. Note that the `initialized` simulation global is used to synchronize the router with the `initiateRoutingTable` procedure.

6.8.4. "index"(mutable, indexer, any type): action;

The **indexed assignment** operation assigns the value of an component given a mutable value having that attribute, an enumeration value specifying which attribute is intended, and a value to be assigned. The value must be of the component's type.

Commentary

```
# STATUS: current

L: list := new list;
L := list'["my", "dog", "has", "fleas"];
L[1] := "cat";
confirm L[1] == "cat";
```

Here the indexable component `1` is being assigned the value `cat`.

6.9. Sequences

6.9.1. sequence(t: type): type;

A sequence is an indexable value. Its length may be obtained by means of the **length** attribute.

The index and dot qualified reference operations of 6.7 apply to sequences. If the sequence is mutable, the index and dot qualified assignment operations of 6.8 also apply.

Commentary

A sequence is an indexable value. It is an abstract type that is effectively the supertype of lists.

```
# STATUS: current

square(x: sequence): int is
        return (length x) * (length x);
a: type is list int;
b: type is list string;
A: a := new a;
B: b := new b;
A := a'[5, 4, 3, 2, 1];
B := b'["v", "w", "x", "y", "z"];
confirm (square A) = 25;
confirm (square B) = square A;
```

The parameter `t` is the type of each indexable attribute.

The index operation of 6.7 apply to sequences.

In the following example `B[0]` is an index operation.

```
# STATUS: planned

element(S: sequence, i: indexer): any is
        return S[i];
B: (list string) := list'["v", "w", "x", "y", "z"];
confirm element(list'["v", "w", "x", "y", "z"], 2) == "x";
confirm (length B) = 5;
confirm B[0] == "v";
```

If the sequence is mutable, the index assignment operation of 6.8 also apply.

```
# STATUS: planned

my_assign(S: sequence, i: indexer, val: any): any is
        S[i] := val;
L :: list := list'["v", "w", "x", "y", "z"];
my_assign(L, 3, "abcd");
confirm L[3] == "abcd";
```

6.9.2. catenate(any...): list;

The **catenate** operation returns a list containing all the elements of its arguments in the order given. The length of the result is the sum of the lengths of its arguments. Any argument that is not a sequence is interpreted as a one element list, with its value as the only element (see 3.10).

Commentary

```
# STATUS: current

arthropod: type is enum(crustacean, insect, spider, millipede);

little_list: type is list int;

A :: little_list := new little_list;
B :: little_list := new little_list;
C :: list := new list any;

A := little_list'[1, 2, 3];
B := little_list'[4, 5, 6];

C := catenate(A, B, 99);
confirm C == list'[1, 2, 3, 4, 5, 6, 99];
confirm (length C) = 7;
confirm A[spider] = 3;
```

Note that the length of the result is the sum of the length of its arguments.

The indices of the return value are 0.. length-1. Any argument that is not a sequence is interpreted as a one element list, with its value as the only element.

6.9.3. copy(any): any;

The copy operation returns a copy of its parameter if it is a mutable. If its parameter is immutable, it returns the parameter (i.e. it acts as an identity operator).

There are two special cases: when applied to an actor, **copy** returns a mutable type that is not an actor; when applied to a simulation, **copy** returns a mutable type that is not a simulation. In these two cases, some of the fields are shared, and some are copied.

6.9.4. append(list, list...): action;

The **append** operation appends to the first list argument the items in the remaining lists. After the append, the length of the first list is the sum of the lengths of all the lists.

Commentary

```
# STATUS: current

L: list := new list;
M: list := new list;
N: list := new list;

L := list'[100, 101, "A"];
M := list'[200, 201, "B"];
N := list'[300, 301, "C"];
append(L, M, N);
confirm L == list'[100, 101, "A", 200, 201, "B", 300, 301, "C"];
```

6.9.5. bring_to_front(list, indexer): action;

This operation removes the indexed item of the list and appends it to the list.

6.9.6. send_to_back(list, indexer): action;

This operation removes the indexed item of the list and inserts it at the beginning of the list.

6.9.7. get(any, first: enumeration, last: type_of first): list;

The **get sublist** operation returns a list that has the same values as the first.. last elements of the sequence. The indices of the return value are 0..last-first.

Commentary

The **get sublist** operation returns a list that has the same values as the first.. last elements of the sequence. The indices of the return value are 0.. last-first. This is illustrated in the following example:

```
# STATUS: current

big_list: type is list int;

B :: big_list := new big_list;
L :: list := new list any;

B := big_list'[100, 101, 102, 103, 104, 105, 106, 107, 108, 109];
L := get(B, 2, 5);

confirm L == list'[102, 103, 104, 105];
confirm get(B, 3, 5) == list'[103, 104, 105];
```

6.9.8. string: (sequence ?) type;

A string is a sequence of characters.

Commentary

Strings are sequences of characters; this implies that they may be lists of characters.

```
# STATUS: current

str :: string := "This is a string";
confirm str = "This Ms a string";
```

6.10. Lists

6.10.1. list(t: type): (sequence ?) type;

A list is a variable length mutable sequence. The elements of a list are assignable and are indexed by the integers 0.. length-1. Elements may inserted or removed from a list at any indexer and the length is adjusted accordingly.

Commentary

```
# STATUS: current

name_list: type is list string;
C :: list := new name_list;
push(C, "Peter", "Rodrigo");
confirm C[0] == "Peter";
```

6.10.2. length(sequencelvector): indexer;

The length operator returns the number of elements in the specified list.

Commentary

```
# STATUS: current

name_list: type is list string;
C :: list := new name_list;
push(C, "Peter", "Rodrigo");
confirm (length C) = 2;
```

6.10.3. last(sequencelvector): indexer;

The length operator returns the index of the last element in the specified list; it is the same as length - 1.

Commentary

```
# STATUS: current

name_list: type is list string;
C :: list := new name_list;
push(C, "Peter", "Rodrigo");
confirm (last C) = 1;
```

6.10.4. index_of(list, item: any): indexer;

This operation returns the index of the first occurrence of the item within the list. If the item is not on the list, the length of the list is returned. If the list is a string, the item must be a string of length one.

Commentary

Some examples of using index_of:

```
# STATUS: current

L :: list := new list any;
```

```
push(L, 9, 8, 7, 6);
confirm index_of(L, 8) = 1;

L := list'[99, 44, 3, 101];
confirm index_of(L, 3) = 2;

confirm index_of(list'[3, 1, 5, 2, 9, 7], 9) = 4;

confirm index_of("abcd", "c") = 2;
```

Restrictions

The only immutable list element types supported by the initial beta release are int, token, enum types, and characters. It does not support non-int numbers nor most author defined immutable types. All mutable types are supported.

6.10.5. "member"(all, list): boolean;

The **member** operation returns true iff the first argument is an element of the second argument.

Commentary

Example:

```
# STATUS: current

S:: string := "gargantuan";
L: (list any) := new list any;
L := list'["a", 3, S];
confirm member(S, L);
confirm member(3, L);
```

L: (list any) := new list any; L := list'["a", 3, "gargantuan"]; assert(! member("lillipution", L), "member 1"); assert(! member(6, L), "member 2");

6.10.6. "..."(list): expression;

This operator returns the elements of its parameter as an expression. It is convenient to use with operators that take variable numbers of parameters.

Commentary

```
# STATUS: current

f(i: int, j: int): int is
        return i + j;
l :: list := new list int;
push(l, 3, 9);
confirm (f l...) = 12;
```

Note that if an expanded list is passed to a function with a fixed number of parameters, overload resolution will fail unless the list has the right number of parameters.

```
# STATUS: current

f(i: int, j: int): int is
        return i + j;
l: list int := [3, 9, 7];
confirm f(l...) = 19; # Fails - wrong number of parameters
```

6.10.7. pop(list): (the list).t;

Pop returns the value of the last element of the list, and reduces the length of the list by one. If the list is viewed as a stack or last-in-first-out queue, then this is the stack pop operation. This is a constant time operation.

Commentary

```
    # STATUS: current


    name_list: type is list string;
    C :: name_list := new name_list;
    D :: string := "";

    push(C, "Alice", "Cooper");
    D := pop C;
    confirm D == "Cooper";
    confirm C == list'["Alice"];
```

6.10.8. push(x: list, (any type)...): action;

Push inserts each of its arguments, beyond the first, at the end of the list. The last element, i.e., x[(length x)-1], . If the list is viewed as a stack or last-in-first-out queue, this operation is the stack push operation. This is a constant time operation.

Commentary

```
    # STATUS: current



    name_list: type is list string;
    C :: name_list := new name_list;

    push(C, "Alice", "Cooper", "lives", "in", "Phoenix");
    confirm C == list'["Alice", "Cooper", "lives", "in", "Phoenix"];
```

6.10.9. truncate(x: list, n: indexer): action;

Truncate removes the `n'th` and subsequent elements of the list. If `n` is greater than the length of the original list, then the list is not truncated. This is a constant time operation. The first element in the list is defined to have address "0" .

Commentary

```
    # STATUS: current


    name_list: type is list string;
    C :: name_list := new name_list;

    push(C, "Alice", "Cooper", "lives", "in", "Phoenix");
    truncate(C, 3);
    confirm C == list'["Alice", "Cooper", "lives"];

    output C;
```

6.10.10. insert(x: list, indexer, x.t...): action;

Insert inserts elements into a list before the `n'th` position. It is an error if `n` is greater than the length of the original list.

Commentary

```
# STATUS: current

name_list: type is list string;
C :: name_list := new name_list;

push(C, "Alice", "Cooper", "lives", "in", "Phoenix");
insert(C, 3, "quite");
insert(C, 4, "happily");
confirm C == list'["Alice", "Cooper", "lives", "quite", "happily", "in", "Phoenix"];

fido: string := new string;
fido := "My dog";
insert(fido, 3, "f", "i", "n", "e", " ");
confirm fido == "My fine dog";
```

6.10.11. assign(x: list, indexer, default: any, x.t): action;

Assign replaces the element at the specified index in the list with the specifed value. If `n` is greater than the length of the original list, then the list is expanded using the value `default` before the assignment is done.

Commentary

```
# STATUS: current

name_list: type is list string;
C :: name_list := new name_list;

push(C, "Chocolate", "is");
assign(C, 5, "very", "good");
confirm C == list'["Chocolate", "is", "very", "very", "very", "good"];

s: string := "My dog";
assign(s, 9, "-", "D");
confirm s == "My dog---D";
```

Assign is an efficient and convenient way to initialize large lists. Instead of

```
# STATUS: current

for i: each 1 .. 10000 do
        push(L, 99);
```

it is much cleaner and faster to write

```
# STATUS: current

assign(L, 10000, 99, 99);
```

6.10.12. remove(list, indexer): action;

Remove deletes the `n'th` element of a list. If `n` is greater than the length of the original list, then no deletions are made. This is a constant time operation.

Commentary

```
    # STATUS: current


    name_list: type is list string;
    C :: name_list := new name_list;

    push(C, "Alice", "Cooper", "lives", "happily", "in", "Phoenix");
    remove(C, 3);
    output C;
    confirm C == list'["Alice", "Cooper", "lives", "in", "Phoenix"];
```

6.10.13. remove_items(list, items: list): action;

Remove_items deletes the first occurence of each element in the second list from the first list, or does nothing if the element does not occur in the first list.

Commentary

```
    # STATUS: current


    D:: number := 87;
    L:: list any := list'[101, vector(1, 2, 3), D, 99];
    remove_items(L, list'[99, vector(1, 2, 3)]);
    confirm L == list'[101, D];
    remove_items(L, L);
    confirm L == list'[];

    name_list: type is list string;
    C :: name_list := new name_list;
    push(C, "Alice", "Cooper", "lives", "happily", "in", "Phoenix");
    remove_items(C, list'[C[2], C[3], C[4]]);
    confirm C == list'["Alice", "Cooper", "Phoenix"];
```

6.10.14. fifo_push(x: list, x.t...): action;

Fifo push inserts each of its arguments, beyond the first, at the beginning of the list. The first element, i.e. x.0, of the resulting list will be the last argument. The time for this operation is proportional to the length of the list. If lower cost fifo queueing is required a tree sort 3 algorithm should be used.

Commentary

```
    # STATUS: current


    name_list: type is list string;
    C :: name_list := new name_list;

    push(C, "Alice", "Cooper", "lives", "in", "Phoenix");
    fifo_push(C, "singer", "renowned", "The");
    confirm C == list'["The", "renowned", "singer", "Alice",
            "Cooper", "lives", "in", "Phoenix"];
```

The time for this operation is proportional to the length of the list. If lower cost fifo queueing is required a tree sort 3 algorithm should be used.

6.11. Examples of Lists

Commentary

The following further illustrates the use of lists.

```
# STATUS: current


list_example(): action is
        fruit: type is enum (apple, pineapple, strawberry, banana, pear);

        weekday: type is enum(mon, tues, wed, thu, fri);

        L: type is list any;
        list1 :: L := new L;
        # list1 := [5, 4, 93];
        push(list1, 5, 4, 93);
        push(list1, 1000, 7, 8);
        output list1;
        output "\cr";

        confirm list1 == list'[5, 4, 93, 1000, 7, 8];
        output "calling push\cr";
        push(list1, 3, 5);

        fifo_push(list1, 33, 129);
        confirm list1 == list'[129, 33, 5, 4, 93, 1000, 7, 8, 3, 5];
        output list1;

        output "\cr";

        # Outputs 129, 33, 5, 4, 93, 1000, 7, 8, 3, 5
        for item: each list1 do
                output(item, ", ");
        output "\cr";

        # list1 := [3, "mon", "pear"]; # Makes list immutable
        truncate(list1, 0);
        push(list1, 3, "mon", "pear");
        output list1;
        insert(list1, 2, "fri");
        output list1;
        confirm list1 == list'[3, "mon", "fri", "pear"];

        remove(list1, 2);   # removes third element list 1
        confirm list1 == list'[3, "mon", "pear"];
        output list1;
        truncate(list1, 2); # truncates list to 2 elements
        confirm list1 == list'[3, "mon"];
list_example();
```

6.12. External Attributes

An external attribute is an attribute of a mutable type that is defined and managed separate external to examples of the type. The values of the external attributes for all existing references to examples of the type form a single example of type attribute.

Chapter 7. Operations and Statements

7.1. Defining Operations

An operation is a type describing a relationship among the types of its formal parameters and its return type. An operation may be evaluated or executed to produce a value or state change effect given appropriate actual parameter values. Each such evaluation is an example of the operation type. Operations are a supertype of functions, procedures, actors and simulations.

7.1.1. control "_define"(token, t: lambda type, t.rt): property;

The definition of a operation is a modified form of the define operation as given in 3.2.1 It differs in that the second parameter in the definition of an operation must be a lambda type and the body must include statements. The body of an operation is a description of a class of computations. Each application of an operation creates one example of the class as a evaluation instance of the computation. The return type of the lambda must be a type of the value resulting from evaluation of the body or action is no value results. If the body of an operation is unspecified (see 2.6.1), it indicates only that the definition is incomplete as specified at this point. Syntactically, the formal parameter list, result type, or body may be omitted. If the formal parameter list is omitted, it is taken as a list of length zero. If the type is omitted, it is taken as unspecified, which is equivalent to "all **|** action". If the body is omitted it is either unspecified or specified elsewhere. Every call on define must have an explicit type or body, although **?** may be used in either case.

7.1.2. function: lambda(?,?) type;

Function is the type of all operations that are both side-effect free and return a value. The apply operation of 3.9.1 applies to function. The application of a function is called an expression.

Commentary

```
# STATUS: current

f1(i: int, j: int): int is
        return i + j;
f2(i: int, j: int): int is
        return i * j;

f3: function := f1;
confirm f3(2, 5) = 7;

f3 := f2;
confirm f3(2, 5) = 10;
```

7.2. Side-effects in Expressions

Value returning operations may have side-effects. In general such effects should be rare and should not be visible at the points of call on the operation. Their most common use would be internal diagnostics and debugging such as counting the number of time an operation is called.

The language does not specify the order of multiple side-effects in an expression or statement. When order matters, call only one operation with side-effects per statement. The language does not guarantee that the nominal side-effects of value returning operations will be preserved over all optimizations.

Commentary

Functions are routines that return values. There are many built in functions such as **sin**, **min**, **max**, etc. Function bodies must always contain at least one "return" (see 7.2.3).

Functions are constructed as in the following examples.

```
# STATUS: current


car(finish: pattern): type is
        wheels :: int := 4;

common_color(car1: car, car2: car): boolean is
        return car1.finish = car2.finish;

function_example1(): action is
        f(a: int, b: int): int is
                return a + b;
        confirm f(2, 3) = 5;

function_example2(): action is

        # This is a one-dimensional world
        location: type is number;

        # Function "distance"  returns a value of type "number"
        distance(pt1: location, pt2: location): number is
                return pt1 - pt2;

        # Ants are passed an initial location
        ant(location: number): type is
                name:: string := "Annie";

        # Create 2 ants
        ant1 :: ant := new ant random(uniform, 0.0, 10.0);
        ant2 :: ant := new ant random(uniform, 0.0, 10.0);

        # Check the distance between them
        if distance(ant1.location, ant2.location) < 5.0 then
                output "ant 1 is close to ant 2\cr" ;
        else
                output "ant 1 is not close to ant 2\cr";

confirm common_color(new car white, new car white);
confirm (! common_color(new car blue, new car white));

function_example1();
function_example2();
```

7.2.1. action: type;

Action is the pseudo type of the return value of procedures. Action is roughly equivalent to the void class of the C language.

Commentary

The type `action` has been used in the first line of most runnable examples we have seen - as a mechanism to make them runnable.

The following is another example of the use of an action inside a runnable example.

```
# STATUS: current

action_example(): action is

        # A knight is just a location
        knight():  type is
                x :: int := ?;
                y :: int := ?;

        # Knights move over 1 and up 2
        move(kt: knight): action is
                kt.x := kt.x + 1;
                kt.y := kt.y + 2;

        # Create a knight
        k :: knight := new knight();
        k.x := 3;
        k.y := 4;
        output("Knight's initial location is " ,
                k.x, ", ", k.y, "\cr");

        # Move the knight
        move k;
        output("Knight's new location is " ,
                k.x, ", ", k.y, "\cr");

action_example();
```

7.2.2. procedure: type is lambda(?, action);

Procedure is the type of all operations that do not return a value. The return type of a procedure is always action. Application of a procedure is called a statement.

Commentary

The type **procedure** is the type of all operations that do not return a value. The return type of a procedure is always `action`. The application of a procedure is called a statement.

```
# STATUS: current

resign(): action is
        output "I can't take it any more.";
abdicate :: procedure := resign;
abdicate(); # Prints I can't take it any more
```

In this example the procedure `resign` is defined, then is assigned to the procedure variable `abdicate` . Notice that in

```
abdicate :: procedure := resign;
```

it is the absence of parentheses after resign that indicates resign is not to be called, but rather just assigned to abdicate without being evaluated.

```
# STATUS: planned

x(int, string): action is
        output (the int, the string);

p: procedure := x(3, ?);
p "abc" ; # Prints 3 abc
```

```
PA[0](); # print "hello"
```

Here p is defined to be a procedure whose value is the partial application of x to 3.

The procedure type allows procedures to be computed at runtime, which is frequently useful:

```
# STATUS: current

p1(): action is
        outln "hello";
p2(): action is
        outln "bye";
p: list := list'[p1, p2];
p[0](); # print "hello"
```

Here is an example of functions which return procedures:

```
# STATUS: current

f(a : int) : procedure is
        if a > 0 then
                return f2;
        else
                return f3;

f2(b : int, d : int) : int is
        return b * d;

f3(b : int, d: int) : action is
        return b + d;

confirm (f 1) (10, 20) = 200;
confirm (f 0) (10, 20) = 30;
```

7.2.3. return(any type): action;

Every execution sequence within the body of a function or other value returning operation must include a call on the return procedure of one argument. The actual parameter to return must be an example of the return type of the operation. The effect of return is to return its argument value to the point of call on the operation and terminate evaluation of the operation's body.

7.2.4. return(): action;

Return permanently discontinues execution of the procedure and resumes execution of its caller just beyond the point of call. Return is however implicit at the syntactic end of each procedure body.

Commentary

```
# STATUS: current


double(i: int): int is
        return i + i;

erroneous(int): int is
        return "This is an error"; # Illegal
```

```
confirm double 3 = 6;
```

Here the double returns an int, which is the type specified in its definition, while erroneous returns a string, which is an error since its declared return type is "int" .

Return permanently discontinues execution of the procedure and resumes execution of its caller just beyond the point of call. Every execution sequence within the body of a procedure must include a call on the parameterless return procedure. Return is however implicit at the syntactic end of each procedure body.

In the following, '**return**' exits from the body of a procedure. There is an implicit return (which will never be executed) after the last line.

```
# STATUS: current

tgif(): action is
        day: type is enum(mon, tues, wed, thurs, fri);
        D :: day := rand(mon, fri);
        if day = fri then
                output "Tgif";
                return();
        else
                output "Just another work day";
                return();
        output "You should never get here!";
   # Implicit return here
tgif();
```

7.3. Sequential and Conditional Control

Control structures are evaluated for the side-effects they produce in the state of the system, and generally do not return values. Control structures are defined as procedures. Each application of a procedure is a statement. If a parameterless procedure is called where a statement is required, the actual parameter parentheses may be omitted without altering the interpretation.

Commentary

Easel supports the traditional structural and control rules that are available in most traditional programming languages. Compound statements consisting of multiple embedded statements and with appropriate scoping rules are allowed, as are **if** and **select** (the **case** statement of the C language) constructs.

In Easel, **if** constructs can be embedded within **select** constructs and vice versa. Labeled statements, most often associated with goto statements, are also allowed in the language.

7.3.1. null(): action;

The null procedure is the do nothing operation. It is called without arguments to explicitly indicate that the absence of a statement.

Commentary

```
# STATUS: current

if (3 = 7) then
```

```
            output "Arithmetic broken\cr";
    else
            null();                        # do nothing
    output "Done with test\cr";
```

7.3.2. null((any type)...): action;

Null is also a means of disposing of unwanted values in a statement context. In the latter form it is analogous to a C language function call in a statement context. In the special case of null new t for some type `t`, it is as if there were instead an attribute definition of the form `? :: t := new t`, at least for the purposes of quantified reference, thus allowing quantified reference to the new `t`.

Commentary

Null is also a means of disposing of unwanted values in a statement context. In the latter form it is analogous to a C language function call in a statement context.

```
    # STATUS: current

    f(i: int, j: int): int is
            return i + j;
    null f(3, 8);                        # Discard f's return value
    output "Survived f";
```

In the special case of null new `t` for some type `t`, it is as if there were instead an attribute definition of the form `? :: t := new t`, at least for the purposes of quantified reference, thus allowing quantified reference to the new `t`.

```
    # STATUS: planned

    null new project;
    begin(the project, "Test");
```

This example creates an anonymous project and then references it as `the project`.

Restrictions

Because **the** has not been implemented, it is not currently possible to reference an example created using `null new t`.

7.3.3. "cpnd_stat"(property...): property;

A compound statement allows multiple statements to be combined into a single statement that will be evaluated sequentially in the left to right (top to bottom) order of their appearance. The statements of a compound statement may also include any number of declarations. Any define or attribute definition that appears within a compound statement is both local and private to that compound statement and cannot be referenced from outside. A compound statement defines a visibility scope that is exceptional in that attributes of its parent scope (that are not hidden by local attributes of same name) can be referenced within the compound statement.

Commentary

A compound statement allows multiple statements to be combined into a single statement that will be evaluated sequentially in the left to right (top to bottom) order of their appearance.

```
# STATUS: current

        output "first statement" ;
        output "second statement" ;
        output "third statement" ;
```

This is a compound statement consisting of three calls on the output procedure

The statements of a compound statement may also include any number of declarations.

```
# STATUS: current

output "First statement\cr" ;

        k :: int := 3;
        p(i: int): int is
                return i + 10;
        output "Second statement\cr";
        confirm (p 5) = 15;
        confirm (p k) = 13;

output "Third statement\cr" ;
```

This is a compound statement consisting of three calls on the output procedure and two calls on confirm, but with an attribute definition and a function definition interspersed.

Any attribute definition that appears within a compound statement is both local and private to that compound statement and cannot be referenced from outside.

```
# STATUS: current

p(i: int): action is
        if the int > 0 then
           s :: string := "Greater than zero" ;
        output s; # Error
p 3;
```

Here the **string** `s` is declared in the compound statement of the then branch of the if statement, and may not be referenced outside that compound statement.

A compound statement defines a visibility scope such that attributes of the parent scope (that are not hidden by local attributes of same name) can be referenced within the compound statement.

```
# STATUS: current

test(): action is
        i :: int := 3;
        j :: int := 5;
        confirm i = 3;
        confirm j = 5;

                i :: int := 7;
                k :: int := 9;
                confirm i = 7;
                confirm j = 5;
                confirm k = 9;


test();
```

Here the compound statement declares attributes i and k; notice that the i in test is hidden by the i in the compound statement, but that its j is still visible.

7.3.4. control "if"(any...): action;

The if conditional operation takes an even-length list of conditions and actions. It evaluates the conditions in order, and when it finds one that is true, it executes the corresponding action. If a default action is not specified in the syntax, it is interpreted as null.

Commentary

Note that in the syntax, the parameters to if are written as "if ... else if ... else if ... else".

```
# STATUS: current

if_example(): action is
        i :: int := rand(1, 3);
        j :: int := rand(1, 3);

        if i+j =2 then
                output("i+j = 2");
        else if i+j = 3 then
                output("i+j = 3");
        else if i+j = 4 then
                output("i+j = 4");
        else if i+j = 5 then
                output("i+j = 5");
        else
                output("i+j = 6");
if_example();
```

7.3.5. control cond(boolean, any, any): action;

The conditional operation returns either its second or third actual parameter depending on whether the value of its first parameter is true or false respectively. If the third argument is not specified in the syntax, it is interpreted as null.

Commentary

The cond operator selects between two expressions depending on the value of a condition. For example, `cond(x, a, b)` will evaluate to a if x evaluates to true or probably, and b otherwise.

```
# STATUS: current

confirm cond(true, "good", "bad") == "good";
```

7.4. Iterative Control

7.4.1. control "for"(identifier, any, action, action): action;

A **for** statement causes its third argument to be evaluated the number of times indicated by its first argument. If the third argument is not evaluated at all, then the forth argument is evaluated once. For each evaluation of the third argument, it is surrounded by a compound statement containing a constant attr_def whose name in the first argument to for. The value of the control attribute at each iteration is determined by the second parameter to for. The second parameter consists of one or more the iteration functions **each**, **every** and **reverse** applied to a type. Values of the type are selected one at a time in the order and at the time determined by the iteration function. If no iterator is specified, the third parameter is executed once with the value of the second parameter as the attribute value. If the third argument is not specified in the syntax it is interpreted as null.

7.4.2. "each"(list | (enumeration type) | mutable type): type;

The **each** iterator selects values from its argument one at a time such that each value is selected exactly once. The entire sequence is determined before the first iteration and unaltered even if the value of the actual parameter expression to each changes. The each iterator is similar to the for statement iterator of most modern programming languages. The argument to each must be an enumeration type, an actor or prop type, or a list. Enumeration types are sequenced in the order of the enumeration and lists in the order of their indices. Only references to actors and props created within the current simulation are sequenced, and in the order of their creation. Other argument types may be supported in future versions.

Commentary

Each is a quantifier that selects from a statically determined set of values specified by its argument. The argument may be a list, enumeration type, mutable type, numeric type or other type. For most purposes each is the same as the quantifier, any.

For lists, the set of values are the list's elements. The elements of list are determined once at the application of each and unchanged thereafter.

```
# STATUS: current

L :: list := new list int;
push(L, 1, 3, 5, 7, 9);

for x: each L do
        output(x, " ");
        truncate(L, 1);
output "\cr";
```

This program outputs "1 3 5 7 9" , even though `L` is set to the empty list after the first iteration - this is because the elements to be iterated over are computed prior to execution of the list. (In contrast, if we had used every, the program would have printed `1` .)

For immutable types (including enumeration types, integers and numbers) the set of value are the examples of the type. The type is determined once at the application of each and is not changed thereafter.

```
# STATUS: current

olive: type is enum (jumbo, supreme, enormous, colossal, gigantic);
start :: int := 5;
stop :: int := 10;

for o: each (jumbo .. colossal) do
        output(o, " ");

for i: each (start .. stop) do
        output i;
        start := start - 1;
        if stop < 15 then
                stop := stop + 1;
```

This example prints out `jumbo supreme enormous colossal gigantic 5 6 7 8 9 10` , since the set of examples is computed only once.

The primary use of each is as the second argument to for (7.4.1) where value from the list or type are selected one at a time in list, enumeration or other order.

```
# STATUS: current

each_example1(): action is
    fruits: type is enum(apple, orange, pear, strawberry);
    for f: each (apple..strawberry) do
        output(f, " ");
      # prints "apple orange pear strawberry"
    output "\cr";

each_example2(): action is
    L :: list := new list int;
    i :: int := 1;
    push(L, 3, 6, 9, 12);
    for j: each L do
        confirm (j = i * 3);
        i := i + 1;

each_example1();

each_example2();
```

Restrictions

Iteration over enumeration types (as opposed to enumeration ranges) does not work.

The else branch on for loops has not been implemented.

7.4.3. "every"(boolean |list |(enumeration type)|mutable type): type;

The **every** iterator selects values from its argument one at a time such that each value is selected exactly once. The sequence is determined one at a time preceding each iteration. This means that actions within the body of the for may alter the sequence of values. The every iterator is similar to the while statement inerator of many modern programming languages. The argument to each must be an enumeration type, an actor or prop type, a list, or a boolean. Enumeration types are sequenced in the order of the enumeration and lists in the order of their indices. Only references to actors and props created within the current simulation are sequenced, and in the order of their creation. Booleans cause continued execution as long as the condition is true. Other argument types may be added in future versions.

Commentary

Every is a quantifier that selects from a dynamically determined set of values specified by its argument. The argument may be a list, enumeration type, mutable type, numeric type or other type. For most purposes every is the same as the quantifier, all.

For lists, the set of values are the list's elements. The list is determined once at the application of every, may be shared and thus may change in length or content between selections.

```
# STATUS: current
```

```
    L :: list := new list int;
    push(L, 1, 3, 5, 7, 9);
    for x: every L do
            output x;
            truncate(L, 1);
```

This prints out `1`, because `every L` gets re-evaluated every time through the loop. On the second time around, `L` has been truncated to a single element, which has already been processed, so the iteration terminates.

For enumeration types (including integers) the set of value are the examples of the enumeration. The first element of the enumeration is determined once at the application of every, but the last example is determined dynamically at each selection.

```
    # STATUS: current

    i :: int := 5;
    stop :: int := 10;
    for every (i < stop) do
            output(i, " ");
            i := i + 1;
            if stop < 15 then
                    stop := stop + 1;
```

This will print out "5 6 7 8 9 10 11 12 13 14 ". Note that both "i" and `stop` are modified in the loop, but only the `stop` value affects the range of the iteration.

If the difference between the first and last element is ever less than its value during a previous iteration, the loop stops.

```
    # STATUS: current

    start :: int := 5;
    stop :: int := 10;
    for i: every (start .. stop) do
            output(i, " ");
            if i = 8 then
                    stop := 0;
```

This example prints "5 6 7 8" , because when 8 is reached, `stop` is set to 0, which is less than the previously encountered examples.

For numeric and other types, the set of values are the examples of the type and are determined dynamically at each selection.

Restrictions

For every does not appear to work on enumeration ranges where the bounds are changed within the loop.

7.4.4. reverse(list): list;

The **reverse** operator returns its argument with the elements in reverse order.

Commentary

In the following example, the order of the fruits is inverted.

```
    # STATUS: current
    confirm (reverse list'[45, 99, 3]) == list'[3, 99, 45];
```

7.5. Interpretation Semantics: Implicit Assumptions

Easel allows incomplete and imprecise specifications, intensional references, and computations on quantified types. One consequence is that it is often impossible to complete a computation without obtaining additional information from outside the system or by making assumptions. It is in general impractical to interrogate the user in every case that additional information is needed. Often the user would not know, have ready access to, or be able to obtain accurate answers.

The easel interpreter instead makes assumptions in the following priority:

a. -- Assume a choice that does not lead to an inconsistency over one that does
b. -- In a form x = true where the value of x is unknown, assume x = false
c. -- In a form x = false where the value of x is unknown, assume x = true
d. -- In any conditional control operation assume the else condition if the values of the precondition for the previous choices cannot be determined
e. -- In uses of quantified expressions always make the worst case assumptions

7.6. Interpretation Semantics: Explicit Assumptions

In general, evaluation of programs requires continued maintenance of worst case assumptions for all quantified expressions, recording of all assumptions made by the interpreter (except those that are obvious because all alternatives lead immediately to an illegal execution), non deterministic execution (either by backtracking, sidetracking, or symbolic execution), and a theorem proving mechanism. Although the interpreter's capabilities will improve over time, it will never be able to prove all valid theorems. More importantly, in the absence of complete and precise information, not all questions will be answerable nor all computations feasible from the available information. Instead the interpreter will make assumptions that enable the simulation to proceed, will add the assumptions to types of the current execution to ensure consistent assumptions, and will report the assumptions made to the user for analysis and later incorporation, either positively or negatively, into the program.

7.6.1. control assert(boolean): action;

A point assertion is a property of each instantiation of an active (i.e. functional, procedural, actor, or simulation) type. It asserts that some condition will be true at that point in the activity of each example of the type.

Commentary

A point assertion is a property of each instantiation of an active (i.e. functional, procedural, actor, or simulation) type. It asserts that some condition will be true at that point in the activity of each example of the type. If it is false the program will fail.

```
    # STATUS: current
```

```
assert_example(): action is
        cat: type is
                paws :: int := 4;
        tiger: type is cat;

        Kitty :: tiger := new cat;
        assert Kitty isa tiger;
        output "Test succeeded!";
assert_example();
```

Restrictions

The current implementation simply prints an error message and continues.

7.6.2. control confirm(boolean): action;

Confirm is a utility procedure that tests for a condition and prints "confirmed" if the condition holds. If the condition fails, it is printed followed by "confirmation failed".

Commentary

Confirm is useful to verify that a program is functioning as expected. It is similar to assert, but it is strictly a run-time operation and has no implications for the Easel proof engine.

```
# STATUS: current

confirm (2+2) = 4;
```

7.6.3. failure(): action;

A failure specification marks a point in the execution sequence which should not be reached. In any deterministic evaluation, executing a call on failure will cause the program to terminate with a error report. In a non deterministic program evaluation, as might arise from unresolved quantifiers such as **some**, the interpreter may tentatively try alternative execution sequences depending on which candidates for **some** is involved. If some but not all such paths fail, no error is reported. Instead, the corresponding examples are eliminated as candidates. Failure should be called whenever an inconsistency is detected. Failure is called for all failure conditions in built-in procedures. Built-in functions often return nil instead.

Commentary

A failure specification marks a point in the execution sequence which should not be reached. In any deterministic evaluation, executing a call on failure will cause the program to terminate with a error report.

```
# STATUS: current

output "About to fail...";
failure();
output "Will never get here, because fails first.\cr"
```

In a non deterministic program evaluation, as might arise from unresolved quantifiers such as **some**, the interpreter may tentatively try alternative execution sequences depending on which candidates for **some** is involved. If some but not all such paths fail, no error is reported. Instead, the corresponding examples are eliminated as candidates.

Failure should be called whenever an inconsistency is detected. In the following example, p calls failure when it detects that the upper bound is smaller than the lower bound:

```
# STATUS: current

p(lower: int, upper: int): action is
        if upper < lower then
                failure();
        for i: each (lower .. upper) do
                output(i, " ");
p(10, 3);
```

Failure is called for all failure conditions in built-in procedures. For example, file operations call failure when they detect hardware I/O errors:

```
# STATUS: current

write(f, 123);
```

This call on write will call failure if the operating system reports an I/O error during the file write operation.

Built-in functions often return nil instead of calling failure. For example, the get sublist operator calls failure when it detects that its `start` parameter is larger than its "stop" parameter:

```
# STATUS: current

get("abcdef", 4, 1); # calls failure
```

Restrictions

The deterministic failure operator is implemented, although it does not currently halt the program, but instead simply writes a message to standard error.

7.7. Mobile Code and Expressions as Data

Security and survivability often require that computational descriptions be treated as data. Programs and parts of programs are computed in one node of a network, transferred to another node, and executed there. This ability is required to simulate down loadable code, java applets, viruses, Trojan horses, and other active content. Outside the domains of network security, mission survivability and infrastructure assurance, expression valued data may be used for a variety symbolic computations analysis and synthesis of computer programs, software transformations and optimizations, symbolic mathematics including algebra and calculus, and , program optimization, differentiation, integration and algebraic manipulations.

7.7.1. translate(file_name: string): expression;

The **translate** operation takes a file name and returns the expression represented by the contents of the file.

7.7.2. lex(string): list token;

Calls the Easel lexical analyzer on the specified string.

Commentary

For an example of using **lex** to process a textual data file, see 10.1.6.

7.7.3. parse(list): expression;

Calls the Easel parser on the specified token list.

7.7.4. _s_expression(expression): string;

Returns a string that represents the expression in S-expression form. Used for debugging the parser.

7.7.5. token: enumeration type;

Token is an enumeration type for character strings used as lexical elements in a program.

Commentary

Lexical analysis (see 7.7.2 and A) converts character strings to tokens, which are a more efficient representation for parsing and interpretation.

```
# STATUS: current

token_list: (list token) := lex("my dog loves your dog");
confirm (length token_list) = 5;
```

Here token_list contains tokens for the five strings `my`, `dog`, `loves`, `your`, and `dog`.

7.7.6. identifier: token type;

Identifiers are tokens that are determined at compile time. That is, any actual parameter corresponding to an identifier formal parameter must be a single identifier token syntactically in the program. The actual parameter value will be the token itself.

Commentary

Identifiers in Easel are case sensitive. For example:

```
# STATUS: current

f(n: int): int is
        return n + 2;
F(n: int): int is
        return n + 5;

i :: int := 3;
I :: int := 9;

confirm i = 3;
confirm I = 9;

confirm (f i) = 5;
confirm (f I) = 11;
confirm (F i) = 8;
confirm (F I) = 14;
```

7.7.7. expression(t: type): type #{is (list expression) | any type}#;

The expression type is the type of any program component or mobile code. When an expression takes the form of a list of expressions it the first list item is interpreted as an expression for an operation and the remaining items as expressions for the actual parameters.

Generally, any expression in a program will be evaluated, but by type qualifying it to the expression type (see 3.10.1), the expression itself will be passed as the argument.

7.7.8. control qt(any): expression;

The **qt** operator returns its argument without evaluation.

Commentary

```
# STATUS: current

s: string := format qt 7 + 3;
confirm s == "7+3";
confirm (eval qt 7 + 3) = 10;
```

7.7.9. control quote(any): expression;

Like **qt**, the **quote** operator returns its argument without evaluation, except that it returns definitions as definitions, not as references to definitions.

Commentary

```
# STATUS: current

p(): action is
        outln "hello";
qt_string: string := format qt p;
quote_string: string := format quote p;
confirm qt_string == "p";
confirm quote_string == #"p(): action is
        outln "hello";"#;
```

7.8. Formatting and Printing

7.8.1. format((any type)...): string;

The arguments to format are values of any type including expressions. Format returns the textual image of a program that would translate to the sequence of values. Format will also return the image of individual tokens and literals. For an arbitrary data structure, it will return the type qualified aggregate form that would generate the structure.

Commentary

An example:

```
# STATUS: current

s:: string := "";
n:: number := 10.1;
s := format n;
confirm s == "10.1";
```

7.8.2. output((any type)...): action;

Output converts each of its arguments in order from left to right to a string image and appends the result to the standard output file. Output is a special operation for conveniently outputting to the standard output file. Output works by calling format (see 7.8.1) on std_out and each of its parameters.

7.8.3. outln((any type)...): action;

This is the same as output, except that it writes a newline at the end.

7.8.4. report((any type)...): action;

The **error reporting** operation may be called by the system or by programs to report errors. Each of its arguments in order from left to right are converted to a string image and appended to the standard error file. Report works by calling format on std_err and each of its parameters.

7.8.5. repln((any type)...): action;

This is the same as report, except that it writes a newline at the end.

7.9. Continuously Evaluating Expressions

Commentary

A continuously evaluating expression (CEE) is an expression whose value is a function of time. CEE's may be used to implement dynamic variables whose value is automatically recomputed, as well as to perform continuous interpolation within a simulation.

7.9.1. cee(t: type): mutable type;

The CEE type is the type of expressions whose value is a function of time; their values may be defined incrementally. A newly defined CEE is not defined for any time.

Restrictions

The beta release does not support user-defined **cee**s.

7.9.2. control "var"(any): cee;

Var marks an expression to indicate that the principle operation of that expression is not to be evaluated when var is executed. The parameters to the principle operation, however, will be evaluated.

Var returns an expression for the application of the principle operation to its actual parameter values. Generally, var would be applied to any attribute references that are to vary at each evaluation of a cee that contains them. Var also would be applied to the entire cee expression.

Commentary

The var operator returns a partially evaluated form of the expression it is passed. The expression is evaluated, except that any CEE's and var expressions contained within the expression are left unevaluated.

The var operator is used to delay evaluation of parts of an expression. In the following example, z depends on the dynamic value of x, but not on the dynamic value of y.

```
    # STATUS: broken
```

```
f(i: int): int is
        return i + 100;

var_test(): action is
        x :: int := 3;
        y :: int := 9;
        z :: int := var f((var x) + y);
        confirm (eval z) = 112;
        output("z ", eval z, "\cr");
        x := 100;
        y := 34; # Has no effect on z
        confirm (eval z) = 209;
        output("z ", z, "\cr");
var_test();
```

7.9.3. continuous(a: vector, bso: vector...): cee;

The **continuous** operation creates a new variable that is continuously evaluated using linear interpolation. The **a** parameter is the value of the variable at the time it is created. In addition to **a**, up to three vectors can be passed in: the rate of change **b**, the upper bound of the range **sz**, and the offset **o**.

Continuous variables with positive ranges (**0..sz**) are evaluated using the formula "((a - o + b * delta_t) mod sz) + o", where **delta_t** is the difference between the current time and the time the variable was created. If **sz** is negative, the range is limited to **0 .. +sz** using reflection rather than modulo arithmetic, and switching the sign of **b** whenever an end of the range is encountered.

The evaluation described above is computed independently for each dimension of the vector.

Commentary

```
# STATUS: current

m: unit Length is 1;
s: unit Time is 1;
k_: number is 1.0e3;

demo(): actor type is
        location:: cee := continuous(vector(10 m, 20 m), vector(50 m, 100 m),
                vector(8 k_ m, ~5 k_ m));
        for every true do
                outln value location;
                wait 1 s;
null new(new simulation, demo());
```

This example might be used to control a vehicle that starts at a location 10 m from the left edge and 20 m from the top edge of a field that is 8 km wide and 5 km high, then moves 50 m/s in the x direction, and 100 m/s in the y direction. The vehicle bounces off the boundaries in the y direction, but wraps around in the x direction.

Restrictions

Continuous may not currently be called from the real-time actor.

7.9.4. value(cee, t: number...): vector;

The **value** operation returns the value of the specified continuous variable at the time t. If t is unspecified, it is assumed to be the current time.

Note: **value** is called implicitly if a continuous variable is passed to a primitive operation that requires a vector.

Commentary

```
# STATUS: current

demo(): actor type is
        C:: cee := continuous(vector(10, 10), vector(50, 100), vector(800, ~500), vector(20
        outln value(C, 100);
        outln value(C, 50);
null new(new simulation, demo());
```

7.9.5. set_cee(cee, any...): action;

The **set_cee** operation is the same as the **continuous** operation, except that instead of creating a new continuous variable, it updates an existing one. Passing nil as one of the parameters has the effect of leaving the corresponding value unchanged.

Commentary

```
# STATUS: current

demo(): actor type is
        C:: cee := continuous(vector(10, 10), vector(50, 100), vector(800, ~500), vector(20
        outln value(C, 100);
        outln value(C, 200);
        set_cee(C, value C, vector(20, 50), nil, nil);
        outln value(C, 300);
        outln value(C, 400);
null new(new simulation, demo());
```

Chapter 8. Actors, Neighbors and Simulations

8.1. Actors

8.1.1. actor: mutable type;

An actor is any autonomous participant of a program or simulation. The actors of a simulation are of equal status and are independent of their parents (i.e. an actor may outlive its ancestors).

Actors may have any number of author defined formal parameters and attributes. Each actor also has four private attributes:

```
# the scheduler controlling this actor
sim :: simulation := ?;

# the position of the start of the currently executing frame
lenv :: int := ?;

# the top of the expression stack
tos :: int := ?;

# the top of the mark stack
mstos :: int := ?;

# the next actor in whatever queue the actor is in currently
next :: actor := ?;
```

When an actor is created, it executes in preference to its creator. When the created actor executes the first scheduling operation (wait, wait_mouse, wait_event, delay, when, etc.), or the first requeue, control is returned to the creator.

Commentary

By parent, we mean the actor or subroutine that called **new** on the actor type. There is no explicit relationship between child actors and their parents - e.g. there is no way for a child to find out who its parent is, and there is no way for a parent to find its children unless it keeps track of who they are (by maintaining a list of children, etc.)

```
# STATUS: current


child: actor type is
        for x: each 1..10 do
                wait 30.0;

parent: actor type is
        for each 1..10 do
                null new child;
                wait 1.0;

main(): action is
        s :: simulation := new simulation;
        null new(s, parent);
        wait s;
```

```
main();
```

Here the `child` actors will continue executing long after the parent actor has terminated - notice that each child lives for 300 cycles, while the parent only lives for 10 cycles.

Lists of actors are often used as the control type of an iterative statement (see 7.4).

```
# STATUS: current


family: simulation type is
        members :: list := new list any;

child(n: int): actor type is
        i :: int := n;
        output("--- Newborn child ", n, " has i of ", i, "\cr");
        for x: every sim.members do
                x.i := x.i + 1000;
                output("Now child ", n, " has i of ", i, "\cr");
                wait 3.0;

parent: actor type is
        output "in parent\cr";
        for n: each 1..10 do
                push(sim.members, new(sim, child n));
                wait 1.0;

main(): action is
        s :: family := new family;
        null new(s, parent);
        wait s;
main();
```

Here the child is using `every sim.members` as the control type of a for loop.

8.1.2. new(actor type): the type;

New creates a new actor. This operation is a special case of the new operation of 6.8.3. Each actor is created, its attributes initialized, a reference to the actor is returned to its parent, and only then are evaluation of its parent and of the remainder of its body continued in parallel. Actors are evaluated in parallel with each other in either real or simulated time, and with concurrent or interleaved semantics depending on their scheduling regime.

Commentary

```
# STATUS: current

child(n: int): actor type is
        for j: each 1..10 do
                output("Child " , n, " ", j, "\cr");
                wait 3.0;

parent(n: int): actor type is
        for m: each 1..10 do
                null new(sim, child m);
        for j: each 1..10 do
                output("Parent " , n, j, "\cr");
                wait 3.0;

main(): action is
        s :: simulation := new simulation;
        for n: each 1..10 do
```

```
                null new(s, parent n);
        wait s;
main();
```

In this example, the ten children print out the "Child" lines in parallel with the parent's printing out the "Parent" lines.

Actors are evaluated in parallel with each other in either real or simulated time, and with concurrent or interleaved semantics depending on their scheduling regime.

Actors are initialized prior to return from the call on **new** without the intervening execution of any other actors.

```
# STATUS: current

network: simulation type is
        nodes: list node := new list node;

node(id: int): actor type is
        x:: int := rand(1, 1000);
        requeue();
        for n: each sim.nodes do
                if n.x > 500 then
                        outln "found a far-out node";

N: network := new network;
for i: each 1..5 do
        push(N.nodes, new(N, node i));
wait N;
```

In this example five nodes examine the `x` attributes of the nodes in the network. Without the **requeue** after attribute initialization, there would be no guarantee that the `x` attributes would be initialized before being referenced.

Occasionally more complex initialization is required.

```
# STATUS: current

network: simulation type is
        nodes: list node := new list node;

node(id: int): actor type is
        x::int := rand(1, 1000);
        y:: int := ?;

        if id > 3 & x > 200 then
                y := 0;
        else
                y := 1;

        for n: each sim.nodes do
                if n.y = 0 then
                        outln "found a funny node";

N:: network := new network;
for i: each 1..5 do
        push(N.nodes, new(N, node i));
wait N;
```

This example may not give the intended results, because there is no guarantee that y will be initialized before it is referenced. This can be handled by wrapping the initialization in a function call:

```
# STATUS: current
```

```
network: simulation type is
        nodes: list node := new list node;

initialize(id: int, x: int): int is
        y :: int;
        if id > 3 & x > 200 then
                y := 0;
        else
                y := 1;
        return y;

node(id: int): actor type is
        x:: int := rand(1, 1000);
        y:: int := initialize(id, x);

        for n: each sim.nodes do
                if n.y = 0 then
                        outln "found a funny node";

N: network := new network;
for i: each 1..5 do
        push(N.nodes, new(N, node i));
wait N;
```

Alternatively, explicit synchronization using a **when** operator may be used. This technique is illustrated in the routing table example of 6.8.3.

Note that **new** returns a reference to the actor. This reference can be assigned to an attribute if needed, or discarded by means of the null operation if it is not.

8.1.3. new(simulation, actor type): (the actor) type;

This form of the new operation creates a reference to an actor in the specified simulation. (If the one-parameter form is used, the actor is created in the current simulation.)

Commentary

```
# STATUS: current

boring: actor type is
        for every true do
                output "Yawn\cr";
null new(new simulation, boring);
```

This creates a boring new actor in an anonymous new simulation.

8.1.4. pronoun self: actor;

Any actor may reference itself using the pronoun self. Self always refers to the dynamically current actor.

Commentary

```
# STATUS: current

log(msg: string): action is
        output(msg, ": n = ", self.n, "\cr");

child(n: int): actor type is
        for j: each 1..10 do
                log("Child");
                wait 3.0;

parent(n: int): actor type is
```

```
            for m: each 1..10 do
                    null new(sim, child m);
            for j: each 1..10 do
                    log("Parent");
                    wait 3.0;

    s :: simulation := new simulation;
    for n: each 1..10 do
            null new(s, parent n);
    wait s;
```

Here the `log` procedure uses the self pronoun to access the parameter "n" child and parent actors. Note that these parameters and attributes are not directly visible within `log`.

Restrictions

Reference to the private fields of the actor (**lenv**, **tos**, etc.) has not been implemented.

8.1.5. terminate(): action;

Terminate immediately and permanently discontinue execution of the current actor. Any existing references to the actor will however remain valid. Every execution sequence within the body of an actor must include a call on the parameterless terminate procedure. A call on terminate is however implicit at the syntactic end of each actor body.

Commentary

```
    # STATUS: current

    child(n: int): actor type is
            for every true do
                    output("Child " , n, "\cr");
                    if rand(0, 500) > 400 then
                            output("Child ", n, "terminating.\cr");
                            terminate();
                    wait 3.0;

    parent(n: int): actor type is
            for m: each 1..10 do
                    null new(sim, child m);
            for j: each (1..100) do
                    wait 3.0;

    main(): action is
            s: simulation := new simulation;
            for n: each 1..10 do
                    null new(s, parent n);
            wait s;
    main();
```

In this example, each `parent` creates 10 `children`, each of whom would live forever except for the random check, which will cause them to terminate.

Note that the parent can still access the `children` even after they have terminated, as in the following:

```
    # STATUS: current

    star(n: int): actor type is
            k :: int := n + 10;
            dead :: boolean := false;
            output( "Star ", n, " has been born and now is dead.\cr");
            dead := true;
```

```
main(): action is
        galaxy :: simulation := new simulation;
        s :: star := new(galaxy, star 1);
        wait galaxy;
        confirm s.k = 11;
        confirm s.dead;
main();
```

Here `main` creates a star, which prints a message and then terminates. Even after it terminates, its attributes can be referenced.

Every execution sequence within the body of an actor must include a call on terminate, but there is an implicit call at the syntactic end of each actor body.

```
# STATUS: current

child(n: int): actor type is
   if (n > 999) then
      terminate();
   else
      output n;
   terminate(); # Unnecessary
```

Here the second call on terminate is unnecessary because it is equivalent to the implicit call automatically inserted by the Easel translator.

Commentary

The following example uses most of the constructs described in 8 . It illustrates the use of actor, new, indivisible, terminate, self, wait. It also illustrates the use of can which is defined in 8.2

```
# STATUS: current

river: simulation type is
        v: view := ?;
        fishies: list Salmon := new list Salmon;

# The life cycle of a salmon
states: type is enum(egg, fish, dead);

# Only when a bear is near a salmon can it eat the salmon
# bear & near(it, salmon) can eat(salmon);  # neighbor relation

eat(S: Salmon): action is
        # Set the salmon's state to dead, and its location to off-screen
        S.State := dead;
        S.location := 10000;

        # Increase the eater's satisfaction
        self.satisfaction := self.satisfaction + 2;

# Utility function for moving along the river
move(d: Length): action is
        self.location := self.location + d;
        if self.location > 600 then
                self.location := 600;
        if self.location < 20 then
                self.location := 20;

# Define a salmon's properties
Salmon(): actor type is

        # Initial state is egg
        State :: states := egg;

        # At birth, place the salmon egg at a random
```

```
            # location along the river
            location :: Length := random(uniform, 0.0, 600.0);
            State := fish;

            # Use yellow circles for salmon
            depict(sim.v, var offset_by(paint(circle(0, 0, 5), yellow), var location, 200));

            # In life, the salmon keeps moving until it dies
            for every State != dead do
                    move random(uniform, -5, 5);
                    wait 1;


    # Define a bear's properties
    bear(): actor type is

            # At birth, place the bear at a random location
            # along the river
            location :: Length := random(uniform, 0.0, 600.0);

            # On average, a bear moves 5 m per time interval
            d: Length := 5;

            # The bear's satisfaction is the inverse of its hunger
            satisfaction: int := 0;

            # Use a brown circle for the bear
            depict(sim.v, var offset_by(paint(circle(0, 0, 8), brown), var location, 200));

            # Life cycle of a bear: move up river eating salmon
            for every true do

                    # The bear keeps moving (self = current actor = the bear)
                    move d * random(uniform, 1, 5);

                    # If the bear's not satisfied, he may change direction
                    if satisfaction <= 0 then
                            if rand(1, 20) > 15 then
                                    d := -d;

                    # Satisfaction wanes over time
                    satisfaction := satisfaction - 1;

                    wait 2;

                    # When the bear is near a salmon,
                    # the salmon is eaten
                    # This eat is only legal if the neighbor relation
                    # efined at the top of the program is
                    # satisfied for some salmon
                    for fish: each sim.fishies do
                            if +(location - fish.location) < 5 then
                                    eat fish;

    # Create the river simulation
    Clarion: river := new river;

    # Use a blue line for the river
    Clarion.v := new view(Clarion, "Clarion River", forestgreen, group(paint(polyline(12, 20, 2
    null make_window(Clarion.v, 0);

    # Generate a new bear and 20 new salmon
    null new (Clarion, bear());
    for each 1..20 do
            push(Clarion.fishies, new (Clarion, Salmon()));

    # Wait until the actors in the river are finished
    wait Clarion;
```

8.2. Neighbor Relationships

Neighbor operations are used to describe the relationship among actor and between actors and props. The neighbor operations of this section allow the relationships among actor to be described as operations that can be performed by one actor, the perpetrator, that change the state of another, the victim. By defining such operations within the victim, the can operation may be used to limit which actors may be perpetrators and the private specification to limit which attribute may be referenced. The effect is that there is no global visibility among actors except as explicitly allowed by actor descriptions.

8.3. Simulations

A simulation defines a world of interactions among examples of many different types. It is a belief system about some real or imagined world. Types may be defined local to a simulation and must be self consistent and consistent with other types of that simulation, but need not be consistent with any enclosed, parallel or enclosing simulation. Each simulation constitutes the limit of the visibility scope for all types and operations defined within that simulation. Each simulation has its own simulated time regime, scheduler and simulation controls.

Commentary

Here is an example of creating actors in simulations.

```
# STATUS: current

ant_hill: simulation type is
        ant_count :: int := 0;
ant: actor type is
        sim.ant_count := sim.ant_count + 1;
        for each (1..rand(1, 50)) do
                wait random(uniform, 1.0, 100.0);
        sim.ant_count := sim.ant_count - 1;
main(): action is
        AH :: ant_hill := new ant_hill;
        for each (1..20) do
                null new(AH, ant);
        wait AH;
main();
```

Here `AH` is declared as a new example of an `ant hill` simulation. 20 `ants` are created in the `ant hill`; each ant lives for a random amount of time and then dies. The simulation global `ant_count` keeps track of the number of `ants` currently living in the `ant hill`.

Multiple simulations may be executed concurrently. Here is an example:

```
# STATUS: current

family: simulation type is
        members :: list := new list any;

child(i: int, n: int): actor type is
        output("This is child ", n, " of parent ", i, " \cr");
        wait 5.0;

parent(i: int): actor type is
        for n: every (1..10) do
                push(sim.members, new(sim, child(i, n)));
                wait 1.0;

main(): action is
        myfamily :: family := new family;
        yourfamily :: family := new family;
        null new(myfamily, parent 100);
        null new(yourfamily, parent 200);
```

```
            wait myfamily;
            wait yourfamily;

    main();
```

Here we are running two independent simulations, `myfamily` and `yourfamily`, in parallel. Each simulation creates 10 children. In the output, the trace messages from both simulations will be interleaved.

8.3.1.
scheduler: type is

scale:: scale_type := uniform_scale;
update_interval :: number := 0.1;
speed :: number := infinity;
clock :: number := 0.0;
_anchor_time :: number := 0.0;
_time_q :: all := ?;
_cond_q :: actor := ?;
_wait_sim_q :: actor := ?;
_runq_count :: int := nti 0;
_blocked_count :: int := nti 0;
_conditions_checked :: int := nti 0;
_last_update_tick :: number := 0.0;
_actor_count :: int := nti 0;

Schedulers are used to control simulations. Their fields may be referenced through the `skdr` attribute of simulations (see 8.3.15).

8.3.2. update_interval :: number := 0.1;

The update_interval attribute is a simulation parameter that controls the interval between graphic updates. (To be precise, it is the minimum time between the end of one update and the beginning of the next.) To increase simulation performance, this attribute may be set to values greater than 0.0. The unit is seconds.

Commentary

In the following example, the depictions within the orbiter simulation will be updated only every 5 seconds.

```
    # STATUS: current

    orbiter :: simulation := new simulation;
    (orbiter.skdr).update_interval := 5.0;
    confirm (orbiter.skdr).update_interval = 5.0;
```

8.3.3. speed :: number := infinity;

The speed attribute is a simulation parameter that controls the real time speed of the simulation. Speed is an upper bound on the ratio between simulated and real time. Speed affects the depiction but not the functional results of a simulation. The speed is initialized to infinity which guarantees that the simulation will run as fast as possible within the processing capabilities of the system.

Commentary

In the following example, the speed of the simulation is set to 0.5, ensuring that every second of simulated time will take at least two seconds of real time. In other words, the simulation will unfold in slow motion at half the speed of the events being simulated (assuming adequate processing power).

```
# STATUS: current

orbiter :: simulation := new simulation;
(orbiter.skdr).speed := 5.0;
confirm (orbiter.skdr).speed = 5.0;
```

8.3.4. _clock :: Time := ?;

Clock is an attribute of every simulation. The value of clock is the amount of simulated time between the start of the simulation and the current simulated time. Outside of a simulation, clock has the same value as the real time clock (see 8.3.13).

Commentary

```
# STATUS: current

a(): actor type is
        for every true do
                wait 10.0;
                output(sim.skdr.clock, "\cr");       #    prints 10, 20 ...

null new(new simulation, a());
```

Note: this attribute is for internal use only; authors should access the clock using the clock() function (see 8.3.12).

8.3.5. _anchor_time :: Time := ?;

Anchor_time is an internal attribute used to implement simulation scaling and speed. It is nominally the time at which the simulation started, but adjusted depending for speed and scaling. If the speed is always 1.0, and the scale is absolute scale, anchor_time will always be the starting time.

Commentary

```
# STATUS: current

a(): actor type is
        for every 1..3 do
                wait 10.0;
                output(sim.skdr._anchor_time, "\cr");

null new(new simulation, a());
```

8.3.6. _time_q :: all := ?;

The time queue is a queue of all actors waiting for a time to elapse (see 8.6.2).

8.3.7. _cond_q :: actor := ?;

The time queue is a queue of all actors waiting for specified conditions to become true (see 8.6.5).

8.3.8. _wait_sim_q :: actor := ?;

The simulation wait queue is a queue of all actors waiting for a given simulation to terminate (see 8.7.1).

8.3.9. _conditions_checked :: int := 0;

This is an internal attribute used to implement waitin on simulations.

8.3.10. _last_update_tick :: number := 0.0;

This is an internal attribute used to implement update_interval.

8.3.11. _actor_count :: int := nti 0;

This is an internal attribute used to implement waiting on simulations.

8.3.12. clock(): number;

The clock function returns the current simulation time, or the real time when the caller is not within a simulation. The clock cycles approximately once every 77.67 hours (i.e., every 2^23 1/60th's of a second).

Commentary

```
# STATUS: current

a(): actor type is
        for every true do
                wait 10.0;
                output(clock(), "\cr");       #    prints 10, 20 ...

null new(new simulation, a());
```

8.3.13. rtc(): Time;

The real time clock function returns the time in seconds since the Easel system started. The same operations apply to real time as to simulated time (see 8.5). The real time clock cycles approximately once every 77.67 hours (i.e., every 2^23 1/60th's of a second).

8.3.14. set_speed(number): action;

The **set_speed** operator sets the simulation speed to the specified value and re-computes the anchor of the current simulation to ensure that the relationships among the simulation clock, the real-time clock, and the simulation speed are maintained.

Commentary

If the simulation clock is ever assigned to, **set_speed** should be called with the current speed so that the anchor is reset properly:

```
# STATUS: current

sim.skdr.clock := 100;
set_speed(sim.skdr.speed); # adjust the achor, etc.
```

8.3.15. simulation: mutable type is skdr :: scheduler := new scheduler;

Each simulation is an actor of the program or parent simulation. The specified return time of a simulation must be "simulation". Actors within a simulation have concurrent semantics.

8.3.16. new(simulation type): the type;

New creates a new simulation.

Commentary

See 8.1.3 for an example of calling new on a simulation type.

8.3.17. pronoun sim: simulation;

Sim (see 8.3.17) is a pronoun used to reference the most global scope within the current simulation. Both the built-in simulation scheduler (the attribute `skdr`) and author-defined attributes of a particular simulation type may be referenced by dot qualification on **sim**.

8.4. Belief Systems: Visibility of Simulations

Defines made anywhere within a simulation type are directly visible in a scope that is global to the body of the simulation, and indirectly visible to the actors within that simulation through the use of the simulation pronoun (see 8.3.17).

Each simulation scope hides all attributes and defines of all more global simulations, so that each simulation acts as an independent belief system. All system level definitions, including built-in language defines, however, are visible throughout every simulation. Language defined attributes are visible in any scope in which it is not hidden by a attribute or define of the same name.

Commentary

```
# STATUS: current

corporation: simulation type is
        employees :: int := 40000;
        virtue: type is enum(market_share, innovation, social_welfare);
        greatest_good :: virtue := ?;

        output("simulation started with ", employees, "\cr");

company: actor type is
        if sim.greatest_good = market_share then
                output "I'm a money grubber\cr";
        else if sim.greatest_good = social_welfare then
                output "I'm a tree hugger\cr";
        else if sim.greatest_good = innovation then
                output "I love newness\cr";
Macro_soft :: corporation := new corporation;
Macro_soft.greatest_good := market_share;
Werd :: company := new (Macro_soft, company);
```

```
Peaches :: corporation := new corporation;
Peaches.greatest_good := innovation;
Pipod :: company := new (Peaches, company);
wait Macro_soft;
wait Peaches;
```

Here we model the belief systems of two corporations. The belief that market share is the greatest good is shared by the `Macrosoft Corporation` and by its `Werd` subsidiary, while the conflicting belief that innovation is the greatest good is shared by `Peaches` and `Pipod`.

Here is another example:

```
# STATUS: current

cheesy: simulation type is
        cheese: type is enum(swiss, american, gorgonzola, cheddar, parmesan);
        variety :: cheese := rand(swiss, parmesan);
        max_time :: number := random(uniform, 0.0, 10.0);

cheese_lover():  actor type is
        for every true do
                # wait sim.max_time;
                wait 1.0;
                output("I believe the moon is made out of " ,
                        sim.variety, " cheese\cr");

main(): action is
        s :: simulation := ?;
        for each (1..4) do
                s := new cheesy;
                for each 1..10 do
                        null new(s, cheese_lover());
        wait s;

main();
```

In the above example, `cheese`, `variety` and `max_time` are local to each of the simulations. Within each of the 10 cheesy simulations, all the `cheese_lovers` share a single belief about the composition of the moon.

8.5. Simulated Time

Time is a base dimension of any simulation or description of the real world. Time may be used to measure either duration or absolute time since the beginning of the simulation. See 2.10.4 for the definition of time.

8.6. Scheduling

8.6.1. control "take"(Time, action): action;

Take specifies the amount of time required to execute an action. Within a simulation this becomes the actual time. Outside a simulation take has no effect other than to report the amount of time *actually* taken.

Commentary

The "take" operation leaves unspecified exactly where time is spent during the action.

```
# STATUS: current

run_one_km(): action is
        self.remaining := self.remaining - 1000.0;

ten_k_runner(id: int): actor type is
        remaining :: number := 10000.0;
        for every (remaining > 0.0) do
                take 4.0 to
                        output("Runner ", id, "has ", remaining, " left to run.\cr");
                        run_one_km();

take_example(): action is
        s :: simulation := new simulation;

        for i: each (1..40) do
                null new(s, ten_k_runner i);

take_example();
```

Here each `runner` runs a kilometer in 4.0 time units; nothing is said about how those 4.0 time units are distributed over the running of the kilometer.

8.6.2. wait(interval: Time): action;

The **wait interval** operation suspends the current actor's execution for the specified interval. Wait works in either simulated or real-time actors.

Commentary

Normally, waits should always be used in preference to delays, so that code may be moved from real-time actors to simulation actors and vice versa.

Unlike the **take** operation, the **wait interval** operation specifies that nothing is going on within the actor during the wait.

```
# STATUS: current

ten_k_runner(id: int): actor type is
        remaining :: number := 10000.0;
        for every (remaining > 0.0) do
                output("Runner ", id, "has ", remaining, " left to run.\cr");
                remaining := remaining - 1000.0;
                wait 4.0;

wait_interval_example(): action is
        s :: simulation := new simulation;
        for i: each (1..40) do
                null new(s, ten_k_runner i);
        wait s;

wait_interval_example();
```

Here each `runner` runs a kilometer in zero time, then waits for 4.0 time units.

8.6.3. delay(Time): action;

The **delay** operation is a real-time delay that suspends the current actor's execution for the specified real-time interval.

Commentary

The only purpose of delays is to provide real-time delays in simulated actors. In real-time actors, delays are the same as waits, but waits are preferred so that code can be moved from real-time actors to simulated actors and vice versa.

```
# STATUS: current

ten_k_runner(id: int): actor type is
        remaining :: number := 10000.0;
        for every (remaining > 0.0) do
                output("Runner ", id, "has ", remaining, " left to run.\cr");
                remaining := remaining - 1000.0;
                blocked delay 4.0;

delay_interval_example(): action is
        s :: simulation := new simulation;
        for i: each (1..40) do
                null new(s, ten_k_runner i);
        wait s;

delay_interval_example();
```

8.6.4. until(Time): Time;

The **time until** operation converts absolute time to a duration from the current time. The time until operation might be called in the argument to a wait interval operation to wait until an absolute time.

Commentary

```
# STATUS: current

ten_k_runner(): actor type is
        remaining: number := 10000.0;
        for every (remaining > 0.0) do
                outln("At ", clock(), ", ", remaining, " remaining");
                remaining := remaining - 100.0;
                wait until clock() + 4.0;

s: simulation := new simulation;
for each (1..4) do
        null new(s, ten_k_runner());
wait s;
```

8.6.5. control when(boolean): action;

The **when** operation suspends the current actor's execution until the condition becomes true. If the value of the boolean changes from false to true and back to false between clock increments the true condition may or may not be detected by the when. If the condition is initially true, the actor will not be suspended, but at the implementation level will be requeued fifo-by-priority.

Commentary

```
# STATUS: current

race: simulation type is
        start :: boolean := false;

starter(): actor type is
        wait 110.0;
        sim.start := true;

racer(id: int): actor type is
        for each 1 .. rand(1, 100)  do
                output("Racer ", id, " is getting ready\cr");
```

```
          when sim.start;
          output("Racer ", id, " is starting");

    R :: race := new race;
    null new(R, starter());
    for i: each 1..10 do
          null new(R, racer i);
    wait R;
```

In this example, even though the ten racers take varying amounts of time to prepare for the race, they all start at the same time because they wait for the starter to signal the start of the race.

8.6.6. requeue(): action;

Moves the current actor to the end of the run queue.

8.6.7. control blocked(action): action;

The **blocked** operator specifies that its parameter is to be executed as a blocked operation - that is, that it should prevent the simulation time from advancing.

Commentary

Consider the following program:

```
    # STATUS: current

    include "::Libraries:Metric.easel";
    procrastinator(): actor type is
          for every 1 .. 3 do
                outln("Procrastinator procrastinating");
                delay 5.0 s;
                wait 0.0 s;
    ant(): actor type is
          for each 1 .. 10 do
                outln("Ant delaying");
                wait 0.0 s;
          outln "Ant done!!";

    blocked_test:: simulation := new simulation;
    null new (blocked_test, procrastinator());
    null new(blocked_test, ant());
    wait blocked_test;
```

In this program, it will take about 15 seconds for the ant to finish, because the procrastinator is executing 3 delays of 5 seconds each. Since **delay** is a blocking operation, the ant cannot continue processing until the delays are completed. Contrast that with the following:

```
    # STATUS: current

    include "::Libraries:Metric.easel";
    procrastinator(): actor type is
          for every 1 .. 3 do
                outln("Procrastinator procrastinating");
                unblocked delay 5.0 s;
                wait 0.0 s;
    ant(): actor type is
          for each 1 .. 10 do
                outln("Ant delaying");
                wait 0.0 s;
          outln "Ant done!!";

    blocked_test: simulation := new simulation;
    null new (blocked_test, procrastinator());
    null new(blocked_test, ant());
    wait blocked_test;
```

Here, because the procrastinator does **unblocked** delays, the ant can continue processing, and so finishes almost immediately.

8.7. Observers and Facilitators

Observers are special actors whose activities are not bound by private and neighbor specifications. An observer may be used to collect information, interrogate simulations, or save simulation states. An observer can reference any attribute and apply any function without regard to private and neighbor restrictions.

Facilitators are special actors whose activities are not bound by private and neighbor specifications. A facilitator may be used to initialize and control simulations. A facilitator has the capabilities of an observer but may also assign to any attribute without regard to private and neighbor restrictions. The body of each simulation is a facilitator with respect to that and all embedded simulations. The body of a program is a facilitator with respect to all simulations of that program.

Commentary

Facilitators and Observers are conceptual entities; nothing special needs to be done to create them in Easel programs.

8.7.1. wait(simulation): action;

Only observers can wait for simulations to complete. Wait simulation suspends the simulation until all actors of the simulation are terminated.

8.8. Events

Commentary

Here is a simple producer-consumer example implemented using events.

```
# STATUS: current


Economy(): simulation type is
        product: type;
        warehouse: event product := new event product;

Producer(): actor type is
        for every true do
                post(sim.warehouse, new product);
                wait random(uniform, 1.0, 100.0);

Consumer(): actor type is
        for every true do
                output("Retrieved ", wait sim.warehouse, " from warehouse.\cr");
                wait random(uniform, 1.0, 100.0);

Main(): action is
        e :: Economy := new Economy;
        null new(e, Producer());
        null new(e, Consumer());
```

8.8.1. event: type is list ;

An event is something that happens, an occurrence. An event may occur any number of times. Each occurrence of an event may have associated state information of type `t`.

8.8.2. new(event type): event;

New returns a reference to a new example of the event type. It does not create any occurrences of the event.

8.8.3. post(e: event, e.t): action;

One occurrence of an event is indicated by each call on post. Posting indicates that the event occurred at the current time. The parameter to post is the event information. Posting does not cause waiting. System defined events or posted by the system and do not generally occur in application programs.

8.8.4. wait(e: event): e.t;

Waiting on an event immediately returns the event state information for the earliest occurrence of that event that has not yet been processed (i.e., by a call on wait or get). If all previous occurrences have been processed, waiting suspends the current actor until an occurrence of the event is posted.

8.8.5. get(e: event): (e.t | nil) ;

Get event is similar to wait except that it returns immediately with value nil when no occurrence of the event is pending.

8.9. OS Events

8.9.1. os_event: immutable type

The type **os_event** is an internal, low-level mechanism for handling Operating System events such as mouse clicks and for manipulating scheduler queues. It is not intended for use at the user level.

8.9.2. event_kind: type is enum(update_event, run_event, mouse_event, print_event);

An internal type used to identify the class of os_event being handled.

8.9.3. wait_event(event_kind, time_out: Time): action;

An internal operator used to queue the actor that calls it to wait on the event_queue of that type until an os_event of the type occurs or the time_out time expires.

8.10. Context Navigation Operations

```
scope: mutable type;                 # visibility scopes
```

Scope is the supertype of all visibility scopes.

```
local(scope): scope;               # most local scope of actor
caller(scope): scope;              # caller's scope
```

```
eval(cee, scope): action |any type;  # eval qt in given scope
```

Chapter 9. Dynamic Graphic Depictions

Authors may define any number of views for an application or simulation. Actors individually control their depiction within each view. Users create windows and associate them with views. Depictions are normally dynamic values that vary during the course of a simulation. Depictions may be comprised from polygons, computed shapes, icons, pictures, text, and projectors and portals.

Commentary

Easel provides significant support for graphical depiction of simulation results. The following is an overview of the concepts behind Easel' s graphical support.

The four key concepts are views, drawings, depictions, and windows. A view is the static background against which the actors of a simulation perform. A drawing is any image that can be displayed in a view. A depiction is an attribute of an actor that specifies what drawing in a view is to be done based on the actor's actions. Finally, a window is a rectangular portion of the monitor's screen that has a frame and an interior; the interior is the space through which a user can watch a particular portion of a view.

As an example, consider watching a watching the passing scenery from a car. The windshield provides one "window" into this view. This window does not show the whole view - just a portion of it. The rear window provides another perspective on the view. There maybe other independent windows - for example the dashboard might constitute a separate window. This window may be indirectly connected to the scenery view since the speed of the car is reflected both in the speedometer and the rapidity of change of the scenery view in one of the windows.

Authors developing simulations may define any number of views for those simulations. Users running simulations cannot create views, but they can create windows and associate them with views. Actors within a simulation individually control their depiction within each view.

Thus a simulated cow (an actor) may be depicted within the scenery view. Depictions are dynamic values that vary during the course of a simulation. Depictions may be built up from polygons, computed shapes, icons, pictures, text, and projectors and portals. These elements are described in more detail below.

9.1. Windows, Views and Depictions

Windows are generally in the province of users and are not manipulated directly by programs. Users may create any number of windows per view (including zero). Users control the size and location of the Window on the screen, and the position and magnification of the view within the window.

Windows are created with the view origin in the upper left corner of the window (i.e., in the positive quadrant of the drawing space) and with magnification equal one.

A view is a perspective or way of viewing a simulation. A view includes everything that can be seen from that perspective. Mechanistically, each view has a title, a background and any number of static drawings of props and dynamic depictions of actors within the simulation.

Each view may be treated as an unbounded flat two dimensional drawing space. Views are implemented as points in a drawing space of approximately 2^64 distinct points. Coordinates are pairs of 32 bit floating point numbers giving 32 bit precision for any value in the range 1e-38.. 3.38.

Depictions connect dynamic drawings to views. Actors may explicitly change depictions or define them to have dynamic values. Changes that occur to a depiction between simulated clock intervals are not rendered until completion of that clock interval, thus ensuring that a consistent view is always displayed. Depictions are generally the values of attributes.

9.1.1. view(simulation, title: string, background: pattern, object: drawing): mutable type;

Each view has a title that is displayed in the view menu and as the title of any window displaying the view. It has a content list and a background pattern that are displayed in each window of the view. Typically, the content list would contain static background drawings or be an empty list ("[ ]"). Both static and dynamic drawings may be added later using the depiction operation of . Each view also has the following two private attributes: the simulation to which the view belongs, and a list of views referenced by projectors and portals of the view.

Commentary

A view is a perspective on a simulation. A view includes everything that can be seen from that perspective. Each view is an infinite (to the limits of the floating point representation) two-dimensional drawing space.

Changes that occur to dynamic drawings a during simulated clock interval are not rendered in windows until completion of that interval. Each view belongs to exactly one simulation and is controlled by the clock of that simulation.

```
# STATUS: current


squad_sim: simulation type is
        v :: view := ?;
        soldiers :: list := new list any;

soldier(y: number): actor type is
        x :: number := 350.0;
        depict(sim.v, var offset_by(paint(circle(0.0, 0.0, 3.0), (red)),
                var x, var y));
        for i: each (1..100) do
                x := x + 1.0;
                wait 1.0;

main(): action is
        GI :: soldier := ?;

        # Create the simulation
        s :: squad_sim := new squad_sim;

        # Create the view
        s.v := new view(s, "Squadron", (slategray), nil);
        null make_window(s.v, 0);

        # Create the depictions
        for z: each (0..20) do
                push(s.soldiers, new(s, soldier (z * 10.0)));
        for ms: each s.soldiers do
                depict(s.v, var offset_by(paint(circle(0.0, 0.0, 3.0), (red)),
                        var ms.x, var ms.y));
        wait s;
```

```
    main();
```

This simple simulation shows a line of 20 `navy-blue soldiers` marching from left to right across a slate gray background. Even though each `soldier` is being simulated as a separate asynchronous thread, the `soldiers` in the `squadron` will always appear in the view as a straight line, because updates to the locations of the `soldiers` are not done until the clock is updated. Since all the `soldiers'` depictions are associated with the squadron's clock and view, changes to the drawings will be synchronized.

Each view has a title, a background pattern and a content list. The title is shown in the view menu when the program is run. Selection of an item in the view menu creates a window into the corresponding view. The background pattern fills the view in all regions that do not have drawings. The background may be specified as transparent to implement layers with different transformation properties. The content is a list of all drawings in the view in back to front order. Each view also has a private attributes indicating to which simulation the view belongs and the list of views referenced by projections and portals of the view.

9.1.2. depict(v: view, d: cee | drawing): action;

The **depict** operator adds the specified drawing to the end of the specified view's drawing list. This is the means by which actors add their depictions to the graphics system.

9.1.3. remove_depiction(v: view, cee drawing): action;

The **remove_depiction** operator removes the specified drawing, or all drawings that reference attributes of the specified actor, from the specified view's drawing list. This is the means by which actors remove their depictions from the graphics system.

9.1.4. update_depictions(): action;

The **update_depictions** operator causes the windows containing the views of the current simulation to be re-drawn.

Commentary

Usually authors do not need to concern themselves with the details of updating windows; the system takes care of updates as appropriate, taking into account **update_interval** and so forth. Occasionally, however, it is necessary to force an update immediately, without waiting for the normal update regime to take effect, as for instance to change the color of a button when it is pressed. In these circumstances, the **update_depictions** operator may be used.

9.2. Colors and Patterns

Commentary

Easel provides color constants corresponding to the standard Web colors (see E.1). Authors may define others as appropriate.

9.2.1. rgb(red: indexer, green: indexer, blue: indexer): pattern type;

The **rgb** operation constructs a color from its representation in red-blue-green light based form. For rgb specifications, eight bits are used for each of the primary colors.

Commentary

```
# STATUS: current

D :: drawing := paint(circle(x, y, r), rgb(r, r, r));
```

This example, which assumes that `r <= 255`, fills the `circle` with a shade of gray that is proportional to the radius of the circle, so small circles are very dark, while large circles are almost white.

9.2.2. pattern: type;

A pattern is a type used to paint regions.

Commentary

Patterns can be RGB colors, OpenGL textures, QuickDraw patterns, PICT bitmaps, and so forth.

Restrictions

Only the **rgb** version of patterns has been implemented.

9.2.3. "-"(pattern): pattern;

The **color complement** operation returns its argument with the colors complemented.

Commentary

```
# STATUS: current

confirm (-red) = cyan;
confirm (-rgb(0, 255, 128)) = rgb(255, 0, 127);
```

9.3. Regions

9.3.1. region: immutable type;

Each region is a portion of a view described to the precision of the floating point representation. Three operations are provided for creating regions depending on whether the region is a polygon, polyline or arbitrary shape.

9.3.2. empty_region: region is rectangle(0, 0, 0, 0);

Empty_region is a region constant indicating the region of zero size. The position of the empty region is arbitrary but by convention is always located at the origin.

9.3.3. everywhere_region: region is rgn_comp empty_region;

Everywhere_region is a region constant indicating the infinite region that includes all points.

9.3.4. control rgn_sect(region, region): region;

Returns the region that represents the intersection of the two parameters.

9.3.5. rgn_union(region, region): region;

Returns the region that represents the union of the two parameters.

Commentary

```
# STATUS: planned

D :: drawing := paint(polyline(8, 0, -20, 20) + circle(0, 0, 10)
        + circle(0, 20, 10), red);
```

9.3.6. control rgn_diff(region, region): region;

Returns the region that represents the area not shared by the specified regions.

Commentary

```
# STATUS: planned

D :: drawing := paint(polyline(8, 0, -20, 20) + circle(0, 0, 10) -
        circle(0, 20, 10), red);
```

9.3.7. control rgn_comp(region): region;

The **region complement** operation returns everything outside the given region.

Commentary

```
# STATUS: planned

V: view:= new view(some_sim, "Complement Demo" , yellow,
        paint(-(polyline(8, 0, -20, 0, 20) + polyline(8, -20, 0, 20, 0)), fuchsia));
```

9.3.8. control transform(region, view_transform): region;

The **drawing transformation** operation rotates, scales, and offsets the first argument by the amounts given to form the region that is returned. The transformations are applied in the order given.

Commentary

```
# STATUS: current

Little_antInNorthFacingEast :: region := polygon(0.0, 4.0, 10.0,
        0.0, 0.0, -4.0);
Big_antInSoutheastFacingWest :: region :=
        transform(Little_antInWestFacingEast, 180 degrees, 3, 3, 100, 100);
```

Argument values of zero, one, one, zero, and zero respectively inhibit the corresponding portion of the transformation.

```
# STATUS: current

Little_antInNorthFacingEast: region :=
        polygon(0.0, 4.0, 10.0, 0.0, 0.0, -4.0);
Little_antInNorthFacingWest: region :=
        transform(Little_antInWestFacingEast,180 degrees,1,1,0,0);
Big_antInNorthFacingEast: region :=
        transform(Little_antInWestFacingEast, 0, 2, 2, 0, 0);
```

Negative scales flip the region over both the x and y axes. This operation may similarly be applied to regions to transform the region. NOTE: the initial implementation does not support transformations in or of projections or portals of views that contain text.

9.3.9. control region_of(drawing): region;

The region of operation returns the region of the given drawing. For atomic drawings it is the existing region. For groups the region is computed as the union of the regions of the drawings in the group.

Commentary

```
    # STATUS: planned

    G :: drawing := group(paint(polyline(8, 0, -20, 0, 20), fuchsia),
       paint(polyline(8, -20, 0, 20, 0), lime));
    G2 :: drawing := paint(region_of G, olive);
```

9.4. Lines and Shapes

9.4.1. polygon(Length...): region;

Form a polygon from the list of points given. The last point is assumed to be the same as the first point and thus need not be specified. Polygons of less than three sides cannot be seen.

Commentary

A **polygon** is a region defined by a sequence of three or more points. The points form the vertices of the polygon in the order given (the first point need not be repeated). Two arguments are required for each point (giving the x and y coordinates of the point respectively).

```
    # STATUS: current

    null new view(new simulation, "C",
            yellow, paint(polygon(10,10, 10,20, 20,15), red));
```

Here T is a triangle pointing to the right.

It does not matter whether the points are given in clockwise or counterclockwise order.

```
    # STATUS: current

    s: simulation := new simulation;
    A :: view := new view(s, "A" ,
            yellow, paint(polygon(4,4, 4,10, 10,10, 10,4), red));
    B :: view := new view(s, "B" ,
            yellow, paint(polygon(4,4, 10,4, 10,10, 4,10), red));
```

Both of these views contain a red polygon on a yellow background.

Edges of a polygon cannot cross each other.

9.4.2. rectangle(left: number, top: number, right: number, bottom: number): region;

An internal operation that constructs a rectangle with the specified bounds.

Restrictions

This operator cannot be used in depictions; use a polygon instead.

9.4.3. oval(xa: Length, ya: Length, xb: Length, yb: Length, d: Length): region;

Oval returns an oval region with center points a and b, and with the diameter d equal to the sum of the two radices.

9.4.4. circle(x: Length, y: Length, d: Length): region;

Specifies a **circle** of diameter **d** with its center at **x, y**.

9.4.5. circle(center: vector, d: Length): region;

Specifies a **circle** of diameter **d** with its center at **center**.

9.4.6. polyline(w: Length, vector...): region;

A **polyline** is a region defined by a pen width, w, and a sequence of two or more points. The points may be specified as either two numbers or a 2-dimensional vector.

Commentary

```
# STATUS: current

T :: region := polyline(8, 0, 4, 10, 0, 0, -4);
U :: region := polyline(8, vector(0, 4), vector(10, 0), vector(0, -4));
confirm U == T;
```

Line segments connect each consecutive pair of points to form the polyline. The last point is not connected to the first. The region of a polyline extends w/2 length to the right and left of each polyline segment plus a circle of diameter w around each point.

9.5. Simple Drawings

A drawing is any image that can be drawn in a view space. Most drawings are constructed by painting a region with a color, pattern or view. Icons and pictures may also be used as drawings. Textual drawings are described in 9.7 . Drawing operations create new drawings that are transformations of their arguments. They do not alter the original drawings.

9.5.1. drawing: immutable type;

Drawing is the type of all images that can be drawn in a view.

Commentary

```
# STATUS: current
```

```
     D :: drawing := new drawing;
     D := paint(circle(0.0, 0.0, 5.0), springgreen);
     D := mag(D, 1, 2);
```

In this example the attribute `D` of type drawing is being used first to contain a 5-cm Spring Green circle, and then that same circle scaled to 10 cm in the vertical dimension.

9.5.2. control paint(region, pattern): drawing;

The **paint** operation returns a drawing that is the result of painting the specified region with the specified pattern.

Commentary

```
     # STATUS: current

     D := paint(circle(0.0, 0.0, 2.0), darkorchid);
```

This example paints a 2-cm circle with the color **darkorchid**.

As explained above, the paint operation does not immediately cause the view to be updated; this occurs only at the next clock tick.

9.5.3. control rotate(d: drawing | region, x: Length, y: Length, Angle): drawing;

Constructs a drawing the same as the `d` parameter, except rotated by the **Angle** parameter around an anchor at point [`x`, `y`].

9.5.4. control scale(d: drawing | region, x: Length, y: Length, xmag: Length, ymag: Length): drawing;

The **scale** operation returns a drawing similar to its first parameter but scaled by xmag in the x dimension and ymag in the y dimension. The scaling in both cases is relative to an anchor at point [x,y].

Restrictions

In the beta release, `x` and `y` must have the same value.

9.5.5. control offset_by(d: drawing | region, x: Length, y: Length): drawing;

Constructs a drawing that is the same as the `d` parameter, except offset by the `x` and `y` parameters.

9.5.6. control offset_by(d: drawing | region, offset: vector): drawing;

Constructs a drawing that is the same as the `d` parameter, except offset by the given 2-dimensional vector.

9.5.7. control offset_toward(d: drawing | region, Angle, Length): drawing;

Constructs a drawing that is the same as the `d` parameter, except offset in the specified angle by the specified length.

9.5.8. control clip(drawing, region): drawing;

Constructs a drawing that is the result of clipping the specified drawing with the specified region.

9.5.9. picture(file_name: string): drawing;

The **picture** type creates a drawing from an external resource contained in the specified file.

Commentary

```
# STATUS: current

P :: picture := new picture("Napoleon.pict");
```

This creates a picture of Napoleon.

Restrictions

The only resource type currently supported is the Macintosh PICT format.

9.6. Complex Drawings

9.6.1. control group(drawing...): drawing;

A **group** is a back to front list of drawings. Groups are drawn front to back.

9.6.2. ungroup(drawing): list drawing;

Ungroup returns the drawings from which a group was formed.

9.6.3. view_transform(
rotation: Angle, mag: number,
xoffset: Length, yoffset: Length): immutable type;

A view transform is used in projections and portals to specify how drawing in the view is to be transformed through the projection or portal. Transformations may be applied to view_transforms in the same manner as they apply to any other drawing.

Commentary

Depiction expressions often will involve various transformations on some standard drawing for the actor. The parameters to the transformations would typically include properties of the physical thing being simulated, such as position, heading and facing.

Please consult the Easel demo programs for examples of using view transforms.

9.6.4. control projector(region, view, view_transform): drawing;

Projections are drawings that represent portions of one view within a second view. Projections are dynamic in that whenever the projected view changes, the projection changes accordingly. Transformations applied to a projection apply to both the region and the projected content. Thus the displayed portion of the view remains constant relative to the region, as might occur when a television set is turned upside down or movie projector is turned in a different direction.

Commentary

```
# STATUS: current

V1 :: view := new view(s, "Source", slategray,
        paint(circle 2 cm, khaki));
V2 :: view := new view(s, " Destination" , ivory,
        projector(circle 200 cm, V1, view_transform(0, 3, 5 cm, 7 cm, nil)));
```

This example shows two views being created. The first has a 2-cm khaki circle on a slategray background. The second has an ivory background with a 200-cm projection of the first view in it. The projection shows a 6-cm khaki circle offset 5 cm to the right and 7 cm up - the results of applying the view transformations to the original view.

```
# STATUS: current

battle_sim: simulation type is
        v1 :: view := ?;
        v2 :: view := ?;
soldier(y: number): actor type is
        x :: number := 50.0;
        depict(sim.v1, var offset_by(paint(circle(0.0, 0.0, 6.0),
                (navy)), var x, var y));
        for i: each (1..1000) do
                x := x + 1;
                wait 10.0;

squadron(): action is
        battle :: battle_sim := new battle_sim;
        battle.v1 := new view(battle, "Gray Demo", slategray, nil);
        null make_window(battle.v1, 0);

        d:: drawing := projector(
                rectangle(0.0, 0.0, 300.0, 300.0),
                battle.v1,
                view_transform(0.0, 0.5, 0.0, 0.0));
        battle.v2 := new view(battle, "Azure Demo" , azure, rotate(d, 100.0, 100.0, pi / 4)
        null make_window(battle.v2, 0);

        for y: each (1..20) do
                null new(battle, soldier(20.0 * y));
        wait battle;
squadron();
```

This example creates two views. The first has a slate gray background; the second has an azure background with a half-size 300 x 300 projection of the first view in it. The navy blue soldiers march from left to right across both views. The projection has been rotated 45 degrees relative to the view, and because that rotation applies to the content of the projection, the soldiers are marching at a 45-degree angle as well.

9.6.5. control portal(region, view, view_transform): drawing;

Like projections, portals are drawings that represent portions of one view within a second view, and are dynamic in that whenever the projected view changes, the portal changes accordingly. Unlike the case of projections, transformations applied to a portal apply only to the region, not to the projected content. Thus the displayed portion of the view remains constant constant relative to the projected view, as might occur when one watches a scene through a pair of binoculars.

Commentary

```
# STATUS: current

battle_sim: simulation type is
        v1 :: view := ?;
        v2 :: view := ?;
soldier(y: number): actor type is
        x :: number := 50.0;
        depict(sim.v1, var offset_by(paint(circle(0.0, 0.0, 6.0),
                (navy)), var x, var y));
        for i: each (1..1000) do
                x := x + 1;
                wait 10.0;

squadron(): action is
        battle :: battle_sim := new battle_sim;
        battle.v1 := new view(battle, "Gray Demo", slategray, nil);
        null make_window(battle.v1, 0);

        d:: drawing := portal(
                rectangle(0.0, 0.0, 300.0, 300.0),
                battle.v1,
                view_transform(0.0, 0.5, 0.0, 0.0));
        battle.v2 := new view(battle, "Azure Demo" , azure, rotate(d, 100.0, 100.0, pi / 4)
        null make_window(battle.v2, 0);

        for y: each (1..20) do
                null new(battle, soldier(20.0 * y));
        wait battle;
squadron();
```

This example is identical to the previous one, except that the second view contains a portal rather than a projector. Because the 45-degree rotation does not apply to the contents of the portal, the soldiers are marching in a vertical line across the rotated square.

9.7. Text Layout Regions

Text drawings allow text to be displayed in graphic windows. Each text drawing is a kind of projector into a text document. The text drawing specifies where the text will be displayed, which document is to be displayed, and where to start within the document. The document itself specifies the content to be displayed and its style. The text drawing provide a default style.

Text drawings may be strung together in a sequence such that display of text that overflows one text drawing will be automatically continued in the next text drawing of the sequence. This capability may be used to delineate pages in a document or for layout of text within a page.

9.7.1. frame(region, background: pattern): mutable type is

_text_font :: int := nti 4;
_text_face :: int := nti 0;
_text_size :: int := nti 12;
_text_color :: pattern := (black);

A frame consists of a rectangular region and a background pattern. Each frame also has private attributes specifying its associated layout and the starting index within the layout's string. Transformation operations cannot be applied to drawings containing layouts, but may be applied to their enclosing view through a projection or portal.

Commentary

```
# STATUS: current

D :: drawing := new frame(rectangle(0.0, 0.0, 100.0, 100.0), red);
```

This creates a red square on which text can be drawn.

Restrictions

The beta release does not support transparent frame backgrounds.

9.7.2. style(frame, string, any...): action;

This operation sets the style for the specified frame. The string is interpreted as a Cascading Style Sheet (CSS) specification.

Commentary

Here is an example:

```
# STATUS: current

f :: frame := new frame(rectangle(0, 0, 100, 100), red);
style(f, #"
        font-family: Times New Roman;
        font-size: 24pt;
        font-weight: bold;
        font-style: italic
        color: maroon;
        background-color: yellow;
        "#);
```

This sets the style for the specified frame to maroon 24-point bold italic Times New Roman on a yellow background. For a full example of displaying text, see the next section.

Restrictions

The supported CSS properties are font-family, font-size, font-weight, font-style, text-decoration, color, and background-color.

Font family must be one of the 3 generic font families (serif, sans-serif, or monospace), or a host-specific font name (lists of font names are not supported). Font sizes must be specified in points or millimeters. Font weight must be bold or normal. Font style must be italic or normal. Text decoration must be underline or none. Color and background-color must be either one of the 16 pre-defined HTML colors, or a six-digit CSS hex value (e.g. #80FF80).

Currently frames must be styled *before* being passed to layouts.

9.7.3. layout(body: string, style: string, frame...): drawing;

A text layout region consists of a text body, a style, and one or more frames.

Transformation operations cannot be applied to drawings containing text, but may be applied to their enclosing view through a projection or portal.

Commentary

A layout consists of a string of characters to be displayed, a Cascading Style Sheet description of the text style, and a list of text frames. The string is drawn within the frames using the specified style. Starting at the beginning of the string text is drawn in the first frame.

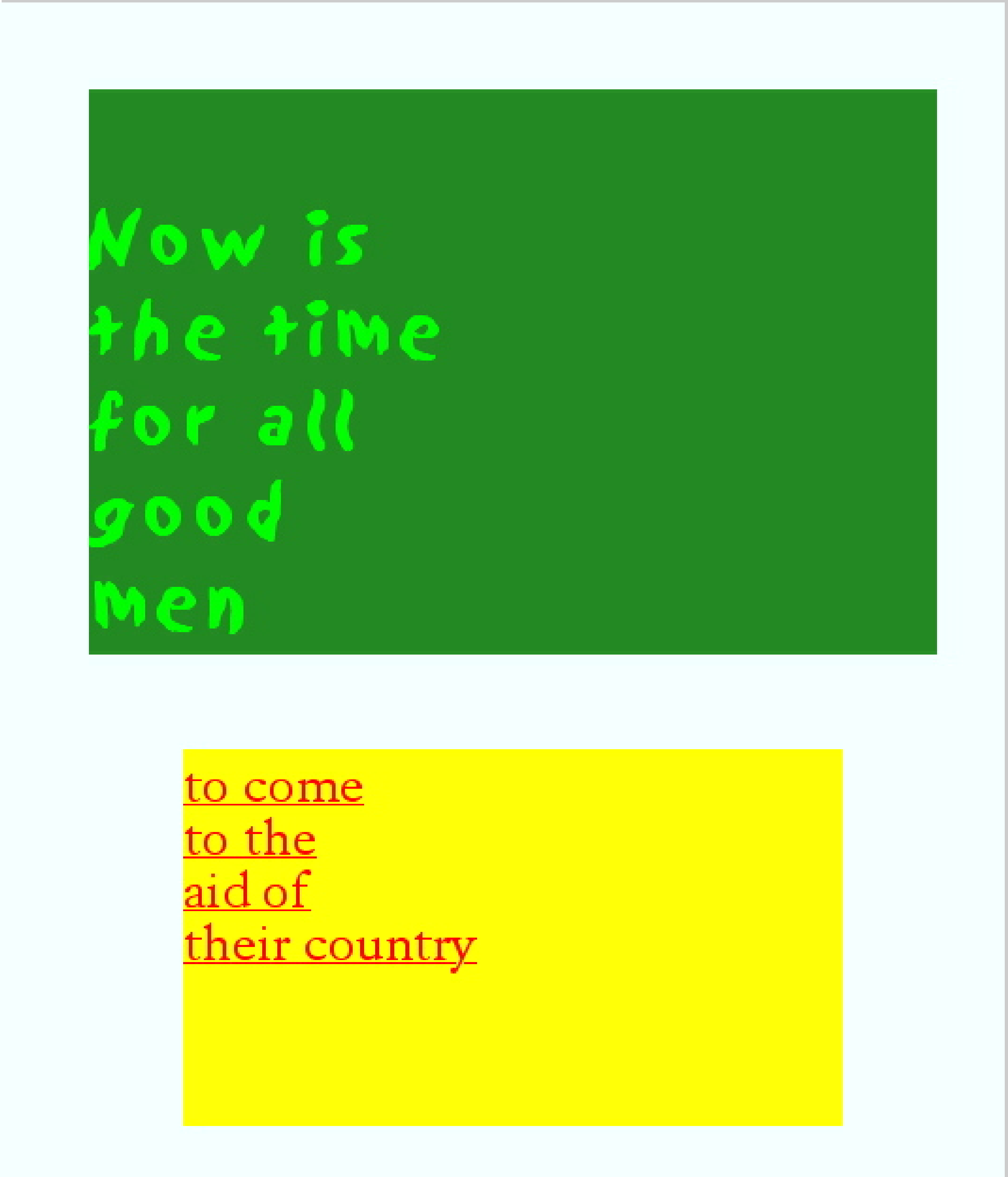
Text lines are folded at word boundaries to ensure that each line is no wider than the frame. When no more lines will fit within a frame, the text is continued at the next frame.

If there are no more frames, the remaining characters are not displayed. A layout appears in drawing objects in the form of the frames from which it is comprised.

Here is an example of displaying text:

```
# STATUS: current

Text_Example(): action is
        v :: view;
        content :: string :=
#"
Now is
the time
for all
good
men
to come
to the
aid of
their country
"#;

        frRect1 :: region := rectangle(50.0, 50.0, 500.0, 350.0);
        frRect2 :: region := rectangle(100.0, 400.0, 450.0, 600.0);
        frame1 :: frame := new frame(frRect1, forestgreen);
        frame2 :: frame := new frame(frRect2, purple);

        style(frame1,
                #"
                font-size: 36pt;
                font-style: normal;
                font-weight: bold;
                color: #00FF00;
                text-decoration: none;
                font-family: Sand;
                "#);

        style(frame2,
                #"
                font-size: 10mm;
                font-style: bold;
                font-weight: normal;
                color: red;
                text-decoration: underline;
                font-family: Hoefler Text;
                background-color: yellow;
                "#);

        v := new view(sim, "Text Example", azure,
                group(layout(content, "", frame1, frame2)));
        null make_window(v, 0);

Text_Example();
```

This creates a layout with two frames. Note that the background-color specification on `frame2` overrides the default specified when the frame is created. Here is the result of executing this program:



Now is
the time
for all
good
men

to come
to the
aid of
their country

Care must be taken when dealing with dynamic texts, as using ""**var layout(... var s ...)**" causes the string to be re-evaluated every time graphics are updated, and this can slow the simulation down. Instead a "**var (var d)**" should be used, where d is a drawing that is assigned a layout, with recomputation of **d** whenever the string changes. See the **Var_Text** demo for an example.

Restrictions

The style parameter is ignored by the beta implementation; style should be attached to each frame. Furthermore, no word wrapping is done - line ends must be inserted manually.

9.8. User Input

User input combines neighbor relations with graphics. The user can be thought of as an actor that can perform certain operations on the state of its neighbors. The operations however are not invoked by calls within the application, but by mouse and keyboard operations in displayed output. The user input operation provide a way of specifying which operations the user may perform, the associated graphic output, how the operations are invoked and how the application will be informed.

9.8.1. get_value(prompt: string, default: string): int;

Displays a dialog box prompting the user for an integer using the specified prompt string and default value.

9.8.2. get_key(): int;

Returns the raw, OS-dependent keycode for the key currently being pressed.

9.8.3. get_char(): string;

Returns the ASCII value of the key currently being pressed.

9.8.4. mouse_op: type is enum (mouse_down, double_click, drag, end_drag, into);

The type of mouse events. Note that the "into" event occurs when the mouse crosses the boundary from one drowing into another.

9.8.5. select_state: type is
op :: mouse_op;
v :: view;
x :: Length;
y :: Length;
m :: int;

The **select_state** type returns information about a given mouse event. The type of event is returned in `op`. The drawing `d` is the drawing that the mouse event occurred over, and the view `v` is the associated view. The coordinates `x` and `y` are relative to the view. The modifier keys that were depressed at the time of the event are encoded in the integer `m`. The **command** key has the value 256, the **shift** key has the value 512, the **shift lock** key has the value 1024, and the **option** key has the value 2048. What is returned in **m** is the sum of the modifier key values.

9.8.6. wait_mouse(mouse_op, view): select_state;

Suspends the current actor until a mouse event of the given type occurs over the given drawing or in the given view. Returns the information about the mouse event.

9.8.7. process_mouse_event(os_event): action;

An internal operation that transfers OS mouse events to the internal event queue.

9.8.8. find_object(view, x: Length, y: Length): list any;

Given a view and a point, the internal operation**find_object** returns the drawing under the point in the view as the last item on the list. (The operation actually walks the drawing tree and returns everything on it, up to the lowest-order drawing that contains the point.)

9.8.9. control selectable(drawing, any): drawing;

Marks the given drawing as selectable and associates the second parameter to it. Usually the second parameter will be the actor that is depicted by the drawing.

Commentary

```
# STATUS: current

slug(v: view): actor type is
        d: drawing :=  selectable(paint(circle(200.0, 200.0, 10.0), slategray), self);
        depict(v, d);
        for every true do
                wait 1.0;
s:: simulation := new simulation;
v:: view := new view(s, "Slug View", azure, nil);
null make_window(v, 0);

null new (s, slug v);
wait s;
```

9.8.10. find_selectable(view, x: Length, y: Length): list any;

The operator **find_selectable** is the same as **find_object**, except that it returns a list which represents objects that have been associated with a drawing by a call on **selectable**.

Commentary

```
# STATUS: current

slug(v: view): actor type is
        C: pattern := slategray;
        d: drawing :=  selectable(paint(circle(200.0, 200.0, 10.0), var C), self);
        depict(v, d);
        for every true do
                wait 1.0;

handler(v: view): actor type is

        ss: select_state := ?;
        hit: list slug := ?;
        b: slug := ?;

        for every true do

                ss := wait_mouse(mouse_down, v);
                hit := find_selectable(v, ss.x, ss.y);

                if (length hit) > 0 then
                        b := hit[0];
                        b.C := rgb(rand(0, 255), rand(0, 255), rand(0, 255));

s:: simulation := new simulation;
v:: view := new view(s, "Slug View", azure, nil);
null make_window(v, 0);

null new (s, slug v);
```

```
    null new handler v;
    wait s;



    # STATUS: current

    hot_sim: simulation type is
            vw :: view;
    color_of(t: indexer): pattern is
            return rgb(min(t, 255), 0, 0);
    hot_bot(): actor type is
            t :: indexer := rand(0, 100);
            d: drawing := selectable( offset_by(
                    var paint(circle(0.0, 0.0, 10.0),
                    color_of var t), rand(0, 500), rand(0, 500)), self);
            depict(sim.vw, d);
            for every true do
                    wait 1.0;

    mouse_handler(v: view): actor type is
            ss :: select_state;
            l:: list hot_bot;
            a :: hot_bot;
            for every true do
                    ss := wait_mouse(mouse_down, v);
                    l := find_selectable(v, ss.x, ss.y);
                    if (length l) > 0 then
                            a := l[0];
                            a.t := a.t + 50.0;

    hbs: hot_sim := new hot_sim;
    hbs.vw := new view(hbs, "Hot Bots", forestgreen, nil);
    null make_window(hbs.vw, 0);
    null 35 new (hbs, hot_bot());
    null 1 new mouse_handler(hbs.vw);
    wait hbs;
```

9.9. 3-D Graphics

The next version of Easel will contain support for 3-D graphics. In this section we present some of the operations that will be available. Users should be aware that the specifications are still under development and subject to change.

Easel 3-D graphics use a different drawing model than its 2-D graphics. Where the 2-D graphics are based on regions, the 3-D graphics are based on filling and stroking paths. Also, the origin for 3-D graphics is at a point in the middle of the screen and 1024 pixels behind it.

9.9.1. path_func: type is enum linear;

The type **path_func** is the type of all supported functions for interpolation between points in paths. The only type supported currently is linear; Bezier and spline are possible future values.

9.9.2. vector_path(path_func, vector...): immutable type;

The type **vector_path** is the type of Easel drawing paths. It consists of a list of vectors representing the points on the path and a parameter that specifies how to interpolate between those points.

Commentary

```
    # STATUS: current
```

```
P: vector_path := vector_path(linear, vector(0, 0), vector(200, 300));
```

In this example, `P` is a path from the origin to the point [200, 300], with linear interpolation.

9.9.3. close_path(vector_path): vector_path;

Returns the specified (potentially open) **vector_path**, but with the path closed.

Commentary

In a closed path, the first and last vectors are the same. If the specified vector is open, **close_path** copies the first vector to the end.

```
# STATUS: current

confirm close_path(vector_path(linear, vector(0, 0), vector(100, 200), vector(100, 100)))
        == vector_path(linear, vector(0, 0), vector(100, 200), vector(100, 100), vector(0,
```

Closing a potentially open vector is required before applying the **fill** operator, since that operator only works on closed paths.

9.9.4. polyline_path(vector...): vector_path;

This operator constructs a polyline from the specified points. It is not automatically closed; it can always be stroked, but can only be filled if the polyline is closed

Commentary

```
# STATUS: current

P: vector_path := polyline_path(vector(100, 100), vector(200, 200));
```

9.9.5. polygon_path(vector...): vector_path;

This operator constructs a polygon from the specified points. The resulting path is automatically closed, so that it can always be either stroked or filled.

Commentary

```
# STATUS: current

P: vector_path := polygon_path(vector(100, 100), vector(200, 200), vector(100, 200));
```

9.9.6. control rotate_3d(obj: drawing | region, axis: vector, theta: Angle): type_of(obj);

Rotates the drawing around the specified access by the specified angle.

9.9.7. control scale_3d(obj: drawing | region, sx: number, sy: number, sz: number): type_of

Returns the drawing scaled by the **sx**, **sy**, and **sz** amounts.

9.9.8. control offset_by_3d(obj: drawing | region, disp: vector): type_of(obj);

Offsets the drawing by the specified displacement.

9.9.9. control offset_toward_3d(obj: drawing | region, dir: vector, dist: Length): type_of(obj)

Offsets the drawing in the specified direction by the specified distance.

9.9.10. stroke(spine: vector_path, shape: vector_path): region;

Returns the region corresponding to the **spine** stroked with the **shape**

Commentary

```
    # STATUS: current
    P: region := stroke(polygon_path(vector(100, 100), vector(200, 200), vector(100, 200)));
```

9.9.11. fill(shape: vector_path): region;

Returns the region corresponding to the surface of the specified vector.

Commentary

```
    # STATUS: current
    P: region := fill(polygon_path(vector(100, 100), vector(200, 200), vector(100, 200)));
```

Chapter 10. Persistent Data

Persistent data is any data that resides beyond the execution of a program either in time or location. Persistent data includes data files, folders and print files.

10.1. Whole File Operations

A mutable file is any file that whose content can be changed. If a value returning operation fails for any reason not explicitly specified in its description, it will report the error and return nil. Failures within non value returning operations will generate an error report. No other error indication is given to the program.

10.1.1. file_format_type: type is enum(data_file, ascii_file, utf_8_file);

Easel supports ASCII text files and data files. In the future, UTF-8 files will also be supported.

Commentary

Files not created by Easel are assumed to be text files.

10.1.2. file(name: string, file_format: file_format_type): mutable type is
name :: string := ?;
val :: any := ?;
os_info :: any := ?;
locked :: boolean := false;
file_spec :: any := ?;

The name of the file is its name in the host OS, and the value is its contents in external storage. The os_info is the file pointer; on the Macintosh it's called the refnum. Locked is the file lock status, and the file_spec is the complete external file specification record (called a fs_spec on the Mac.)

Restrictions

The `locked` boolean is currently unused.

10.1.3. file(file_format: file_format_type): mutable type is
return file(file_format, ?);

Files are data that reside on peripheral storage. Each file has a string name, a storage unit on which it resides, and a content of type t. Files also have a parent file, universal id and a storage relative id as private attributes. References to files are valid whether the file is open or closed. References to existing files are obtained through the folder structure. Assignment to the name parameter of an immutable file (see 6.8.2) changes its name. It is often convenient to categorize files by their content type.

10.1.4. new_file(name: string, file_format: file_format_type): file;

The **new_file** operation creates a new file with the specified name and format. It is an error if the specified file already exists.

10.1.5. make_file(name: string, file_format: file_format_type): file;

the **wake_file** operation creates an empty file with the specified name and format. If the file already exists, it is overwritten.

Make_file is useful when creating a file that may or may not have previously existed, such as in repeatedly debugging or testing of programs.

10.1.6. open_file(name: string, file_format: file_format_type): file;

The **open_file** operation opens a pre-existing file for reading. If the file does not exist, an error is reported.

Commentary

Here is an example of processing a text file using **open_file** and **lex**. The example reads a file, tokenizes it using **lex**, and then computes the sum of all the integers in the file.

```
# STATUS: current

# Open the file
token_list:: list token:= lex read open_file("test.text", ascii_file);

# Compute comma
comma:: token := (lex ",")[0];

# Initialize total
sum:: int := 0;

# Loop through the tokens
for t: each token_list do
        if t isa int then
                sum := sum + t;
        else if t = comma then
                outln "found a comma";
        else
                outln("found ", t);

# Print the total
outln("The sum is ", sum);
```

Checking for any specific token can be done by lexing the appropriate string literal; the example shows how to do this in the case of commas.

10.1.7. read(f: any): any;

Read returns the entire content of file f. The file may be open or closed.

10.1.8. read_URL(URL: string): string;

Read_URL returns the resource referenced by the URL.

Commentary

```
# STATUS: current

confirm (read_URL "http://www.anthus.com/Easel.xml" )
        == "<test><para>Hello Easel User</para></test>\LF";
```

Restrictions

Currently, the behavior of **read_URL** when the specified resource is not text data is undefined.

10.1.9. write(f: any, value: any): action;

Write replaces the entire content of the file by the value. The file must be mutable and may be open or closed.

10.2. Sequential File Operations

A sequential file is any file whose content type is sequence. Each element of the content sequence is a record of the file. At the implementation level easel supports only two kinds of sequential files, those containing sequences of characters and those containing sequences of any type. All foreign files are treated as character files. Records may be rewritten in the middle of the file, and may be added to the end, but insertions in the middle are not currently supported.

10.2.1. file_length(f: any): int;

For a file `f`, **file_length** returns the number of records in the file. For files of type ascii_file or utf_8_file, this is the number of characters in the file; for files of type data_file, it is the number of Easel data objects that have been written to the file.

Commentary

```
# STATUS: current

f: file := make_file("test2.text", ascii_file);
write(f, "12345\cr");
confirm (file_length f) = 6;
```

10.2.2. file_last(f: any): int;

For a file `f`, file_last returns the indexer of the last record in the file. For files of type ascii_file or utf_8_file, this is the number of characters in the file minus `1`; for files of type data_file, it is the number of Easel data objects that have been written to the file minus `1`.

Commentary

```
# STATUS: current

f: file := make_file("test2.text", ascii_file);
write(f, "12345\cr");
confirm (file_last f) = 5;
```

10.2.3. get_element(f: any, num: int): any;

This operation returns the element in the file `f` whose index is num.

Commentary

```
    # STATUS: current

    f: file := make_file("test2.text", ascii_file);
    write(f, "12345\cr");
    confirm get_element(f, 2) == "3";
```

10.2.4. write_element(f: any, num: int, value: any): action;

This operation inserts the given value in the file at the position num.

Commentary

```
    # STATUS: current

    f: file := make_file("test2.text", ascii_file);
    write(f, "12345\cr");
    write_element(f, 2, "x");
    confirm (read f) == "12x45\cr";
```

10.2.5. push_file(f: any, any...): action;

Push_file appends the elements in its second parameter to the file specified by its first parameter.

Commentary

```
    # STATUS: current

    f: file := make_file("test2.text", ascii_file);
    write(f, "12345\cr");
    push_file(f, "67890\cr");
    outln("f is ", read f);
    confirm (read f) = "12345\cr67890\cr";
```

10.2.6. pop_file(f: any): any;

Pop_file returns the last character or data object in the file f and decreases the length of the file by 1.

Commentary

```
    # STATUS: current

    f: file := make_file("test2.text", ascii_file);
    write(f, "12345\cr");
    null pop_file f;
    confirm (pop_file f) == "5";
```

10.2.7. read_line(f: any, i: int): string;

The **read_line** operation reads the next line from file f starting at character position i. If an input-output error is encountered nil is returned. At most 2^15-1 characters can be read at once.

Commentary

```
# STATUS: current

f: file := make_file("test2.text", ascii_file);
k:: int := 0;
s:: string;
t:: string := new string;

write(f, "12345\cr67890\crabcde\crfghij\cr");

for every (k < (file_length f)) do
        s := read_line(f, k);
        t := catenate(t, s);
        k := k + (length s) + 1;

confirm t == "1234567890abcdefghij";
```

10.2.8. append_to_file(f: any, value: any): action;

Append_to_file inserts each item of each sequence at the end of the sequential file. The sequence's items must be of the type required by the file. Append_to_file may be used to append several strings to a text file or printer output.

Commentary

```
# STATUS: current

f: file := make_file("test2.text", ascii_file);

write(f, "line 1\cr");
confirm (read f) == "line 1\cr";

append_to_file(f, "line 2\cr");
confirm(read f) == "line 1\crline 2\cr";

f := open_file("test2.text", ascii_file);
confirm(read f) == "line 1\crline 2\cr";
```

10.2.9. truncate_file(f: any, n: int): action;

The **file truncation** operation removes all records of the file including and beyond record number n. The resulting content will be a list of n items. This operation is analogous to the truncate operation on lists (see 6.10.9). Truncate does nothing if the end of file is less than n.

Commentary

```
# STATUS: current

f: file := make_file("test2.text", ascii_file);
write(f, "12345\cr");
truncate_file(f, 3);
confirm (read f) == "123";
```

Chapter 11. System Level Features

11.1. System Attributes

System attributes are shared among all simulations of a program. They include the random number seed, standard output files, and several real time attributes. There are also private system attributes containing the list of windows, the list of views, the list of enumeration types, and the minimum window update time interval.

11.1.1. version(): string;

Returns the current version number of the Easel system.

11.1.2. sound_mode_type: type is enum(preemptive_sounds, sequential_sounds, parallel_sounds);

The **sound_mode_type** is the type of the system attribute that determines how the play operation (see 11.2.1) behaves if the current sound has not completed when another call on play takes place. With **preemptive_sounds**, the current sound is discontinued. With **sequential_sounds**, the second sound waits for the current sound to finish. Finally, in **parallel_sounds** mode, the two sounds are overlaid one on top of the other.

Restrictions

Sounds that are started in **parallel_sounds** mode will continue to play to the end, even if the mode is subsequently set to preemptive.

11.1.3. registers: mutable type is
std_out :: file;
std_err :: file;
sound_mode :: sound_mode_type := preemptive_sounds;
use_3D_graphics :: boolean := false;
program :: expression;

`Registers` is the type of the registers of the Easel machine. It contains system-wide attributes.

11.1.4. std_out :: file;

The value of std_out is the file to which standard text output is written. Std_out is initialized to the console file. The console window is created automatically the first time the console file is used and thereafter displays the content of the console file. The value of std_out may be referenced and assigned by the program.

11.1.5. std_err :: file;

The value of std_err is the file to which error reports are written. Std_err is initialized to the error reports file. The error report window is created automatically the first time the error reports file is used and thereafter displays the content of the error report file. All errors reported by translator and interpreter are written to std_err. Programs can report errors the same way.

The value of std_err attribute may be referenced and assigned by the program.

11.1.6. sound_mode :: enum(preemptive_sounds, sequential_sounds, parallel_sounds) := preemptive_sounds;

The **sound_mode** attribute determines how the play operation (see 11.2.1) behaves when the current sound has not completed when another call on play takes place. See 11.1.2.

Commentary

The following example shows how to set the system in parallel-sound mode:

```
# STATUS: current
system().sound_mode := parallel_sounds;
```

11.1.7. use_3D_graphics :: boolean := false;

The **use_3D_graphics** attribute determines whether the experimental three-dimensional graphics operations are enabled.

Commentary

The following example shows how to turn on three-dimensional graphics:

```
# STATUS: current
system().use_3D_graphics := true;
```

11.1.8. program: expression;

The **program** register contains the expression for the currently executing program.

Commentary

The following example shows one way to have a program pretty-print itself:

```
# STATUS: current
outln system().program;
```

This prints "outln system().program;", i.e. the program itself.

11.1.9. system(): registers;

The **system()** function returns the current values of the Easel machine registers.

Commentary

Note that system values may be changed by assigning to the result of the function call; for example:

```
# STATUS: current
system().sound_mode := parallel_sounds;
```

11.1.10. current_date(): string;

Returns the current date in a host-dependent format.

11.1.11. current_time(): string;

Returns the current time in a host-dependent format.

11.1.12. current_ticks(): string;

Returns the current host-dependent real-time tick count.

11.2. Sounds

11.2.1. play(sound: string): action;

Plays the sound in the file whose name is passed as the string.

Commentary

The following example plays a purring sound:

```
# STATUS: current

play "::Sounds:Purr.aiff";
```

11.3. Interpreter

11.3.1. eval(e: expression): e.t;

Eval is the interpreter for all expressions of the language. If the expression e was obtained by a var operation, then eval returns the value of e interpreted in the local context of the call on eval. If e was obtained by inhibiting an actual parameter evaluation as specified by a formal parameter of type var, then e is evaluated in the context of that call. In the latter case, eval will fail if evaluation context is not an existing frame of the current actor.

Commentary

The following illustrates the use of eval:

```
# STATUS: planned

mobile_code_example(): action is
   prog :: expression := translate #"
   Code_example(): action is
           str :: string := "qwerty";
   output str;
        #";
        eval prog; # prints "qwerty"
mobile_code_example();
```

Note that this example uses a multiline string delimited by '#"' and '"#'.

11.3.2. reinitialize(): action;

The **reinitialize** operator reinitializes the Easel system. All running simulations are terminated and memory is returned to its initial state.

11.3.3. get_run_count(): int;

The **get_run_count** operator returns a counter that survives over calls on reinitialize. It is initialized to 0 when the Easel system starts up.

Commentary

This operator, in conjunction with **set_run_count**, can be used to run the same simulation with different sets of parameters, as for example when doing Monte Carlo simulations.

11.3.4. set_run_count(int): action;

The **set_run_count** operator assigns a new value to the run_count.

Commentary

This operator, in conjunction with **get_run_count**, can be used to run the same simulation with different sets of parameters, as for example when doing Monte Carlo simulations.

11.3.5. pause(): action;

Pause interrupts the current simulation.

11.3.6. control trace(boolean): action;

Trace turns instruction tracing on and off.

Commentary

After calling `trace true`, a **trace** message will be written in the error file before each each statement is executed. Here is a sample **trace** message:

```
TRACE: print_payroll 18 sum := a + b;
```

Here print_payroll is the name of the containing procedure, 18 is the depth of the stack, and `sum := a + b` is the statement being executed.

Only the first actor to call `trace true` will be traced. Tracing applies only to the level at which **trace** is called, and higher levels.

```
# STATUS: current

main(): action is
        p1(): action is
                p11(): action is
                        report "5\cr";
                trace true;
                report "6\cr";
                p11();
                report "66\cr";
        report "7\cr";
        p1();
        report "8\cr";
        trace false;
main();
```

Here there will be no **trace** for `report 5` because the statements in p11 are lower than the level of the **trace** call in p1. The output from this program will be:

```
    7
    trace p1 14 report 6
    6
    trace p1 14 p11()
    5
    trace p1 14 report 66
    66
    trace main 8 report 8
    8
    trace main 8 trace false
```

Note that the exact values for the stack levels are implementation-dependent.

11.3.7. last_error(): string;

The function last_error returns a string describing the last run-time error that has occurred, or nil if there have been no run-time errors.

11.4. Memory Management

- memory allocation
- pwd(int, int, int): any type;
- garbage collection

11.5. Windows

Windows are updated at most once every sixtieth of a second. Only those portions of a window that has been uncovered or whose content has changed are redrawn. All drawing is front to back with each pixel being drawn only once. All drawing is done with pixel graphic regions for high performance.

11.5.1. window: mutable type;

Although windows could be implemented as projector drawings of the screen, for application level compatibility they are instead implemented as windows of the underlying operating system. Each window is updated whenever portions of it are uncovered or its content changes, but never more frequently than the update interval. Content changes are displayed only at the end of clock intervals of the associated simulation. Only regions of the window that have been changed or uncovered are redrawn.

11.5.2. make_window(view, where: any): window; # where: int|rectangle|vector|any

The **make_window** operation creates a new window on the specified view. The **where** parameter specifies the size and location of the window in screen coordinates; that parameter can be either a rectangle, a length-4 vector with the four values of a rectangle, or an integer value with 1 meaning large and 2 meaning full-screen. Any other value will be interpreted as "normal window size".

Commentary

```
    # STATUS: current

    w: window := make_window(v, rectangle(100, 100, 500, 500));
    w := make_window(v, rectangle 2);
    w := make_window(v, nil);
```

11.5.3. screen_rectangle(): vector;

The **screen_rectangle** operation returns a length-4 vector that represents the effective size of the display, with unusable space such as the menu bar removed. The values are in "top left right bottom" order.

Commentary

```
# STATUS: current
w: window := make_window(v, screen_rectangle());
```

This example creates a window that covers the entire screen.

Appendix A. Lexical Grammar

The lexical grammar specifies the sequences of source program characters that constitute words of the language. The lexical grammar is formally defined using the meta grammar of D.

A.1. Identifiers

Syntax

```
id ::= alpha [{ ["_"] alphanumeric }]
alpha ::= "a"|"b"| . . . "z" |"A"|"B" . . . "Z"
digit ::= "0"|"2"| . . . "9"
alphanumeric ::= alpha | digit
```

Identifiers are used as names to reference values within a program. Upper and lower case characters are distinguished within identifiers.

A.2. Operators

Syntax

```
op ::= "can" |":" |";=" |"is" |"::=" |"take"
        |"&" |"|" |"~" |"!" |"<<" |"==" |">>" |"isa"
        |"<" |"=" |">" |"<=" |"!=" |">="
        |"+" |"-" |"$" |"@" |"*" |"/" |"%" |"\"
        |"^" |"." |"'" |"`"
postop ::= ".." |"..."
```

Operators are used as names of operations. Op's may be used in binary infix form or unary prefix form. Postop's may be used in binary infix form or unary postfix form.

A.3. Numeric Literals

Syntax

```
int_lit ::= digit [{ ["_"] digit}] # integer literal
num_lit ::= int_lit                                     # decimal integer
         | int_lit "." int_lit                    # decimal number
                 [("e"|"E")[("+"|"-")] {digit}]
         | id ["." id] "#" int_lit                   # based number
                 [("e"|"E")[("+"|"-")] {digit}]
```

An integer literal is a sequence of one or more decimal digits interpreted as a decimal number. A numeric literal may be decimal or in any radix from 1.. 36. Numeric literals may have fractional parts and exponent parts. The base or radix of a based number is specified as a decimal integer literal immediately following the "#" which itself must be immediately follow the integer or fractional part. The integer and fractional parts of a based number are composed from the first n characters from the sequence "01234567890ABCE . . . XYZ" where n is the value of the base. In based numbers, alpha digits may be used in either upper or lower case with the same meaning. The exponent is specified as a signed or unsigned decimal integer immediately following the rest of the number. The rest of the number is multiplied by 10 to the power of the exponent value to obtain the final numeric value.

A.4. String Literals

Syntax

```
str_lit ::= '"' str_char '"'
        | '#"' any_char '"#'
str_char ::= printable_char_other_than_dblqt_and_backslant
        | "\" ascii_name | "\" hexnum hexnum
ascii_name ::= "NUL" . . .
hexnum ::= '0' |'1' . . . |"9" |"A" |"B" . . . |"F"
```

String literals have a special lexical form. A string literal is any sequence of printable ASCII characters other than " and \ enclosed by double quotes ('"').

A '\' may appear in a string literal as a control character and followed by as designation of the intended characters. The sequences following '\' may be either an ascii_name or two hexnum's. The ASCII names are any official ASCII name in lower case. The hex digits give the hex value of the intended ASCII character. The resulting string has the actual designated ASCII character replacing the multi character specification.

The second form of string literal allows direct input of arbitrary text without conversion. The string consists of all characters between #" and "#.

Commentary

```
# STATUS: current

output "ASCII values: \\CR = , \CR\\HT = \HT, \\LF = \LF, \\RS = \RS\CR";
output "Hex values: \\50 = \50, \\65 = \65\CR";
output #"This
is
a t\CRest
 "#;
```

This prints out:

```
ASCII values: \cr = ,
\ht =         , \lf = \cr, \rs = \rs
Hex values: \50 = P, \65 = e
This
is
a \crtest of \50 escape sequences.
```

Notice that in multiline string literals, no escape processing is performed.

A.5. Comments

Easel uses # to introduce comments that extend to the end of the line. In addition, Easel uses # to construct balanced delimiters for special purposes. In particular, #{ and }# are used as balanced delimeters for comments that may span multiple lines. As noted in A.4, '#"' and '"#' may be used for uninterpreted string literals.

Commentary

Here is an example that illustrates both kinds of Easel comments.

```
# STATUS: current
```

```
output "abc "; # output "def ";
#{
        output "ghi ";
        #{
        output "jkl ";
        }#
        output "mno ";
}#
output #{ TEST!!! }# "pqr\cr";
output #"This
is
a t\crest
 "#;
```

This prints out "abc pqr". Notice that multiline comments may be nested.

A.6. Fill Space

A.7. Key Words

A.8. Indenting Rules

Each line of a program begins with some sequence of tab and space characters. The amount of white space at the beginning amount determines the number of embedded scope levels of that line within the program. The absolute amount in irrelevant, but the relative amounts of which lines are more embedded. All lines in the same scope should have exactly the same prefix of tabs and spaces. In addition, **{ }** pairs may be used to surround program scopes. Either or both methods may be used for a given scope.

More precisely, in computing the indent amount the lexical analyzer assumes that tabs have much more white space than spaces, it ignores any spaces preceding a tab, and the amount of leading white space on continuation lines (see A.9) does not matter.

A.9. Line Continuation

Any logical line may be folded into multiple physical lines. All but the first physical line of a logical line are called continuation lines. The amount of leading white space on continuation lines does not affect the indenting scope of the logical line. No markers are required to indicate continuation lines in Easel; a carriage return in adequate.

Formally, a continuation line is any line following a line that does not end in one of the following eight words: ..., or that begins with "}" or "{". In determining what begins or ends a line, tabs, spaces, fill characters and comments are ignored.

Appendix B. Semantic Grammar

Syntax

```
prog ::= { prop | stat } [ exp] # easel program
prop ::=   "property" exp        # predicate or supertype property
        |  id ":" type [ "is" body  ] # definition
        |  name [ fpl ] ":" type [ "is" body  ] # lambda definition
        |  name [ fpl ] "is" body # definition with unspecified type
        |  id ":" type ":=" exp         # attribute definition
        |  exp "can" body         # neighbor relationship
        |  id  prop         # computed property
        |  stat         # statement
        |  exp ".." exp ":" type ":=" exp # indexable attribute definition
        |  "attribute" type "." id ":=" exp # external attribute definition
type ::=  exp         # type valued expression
body ::= "{" [{ prop ";" }] "}"                           # body of type definition
        | exp | prop              # simple expression, property or statement as body
name ::=  id | "\q" op "\q"         # name: id or quoted operator
        fpl ::= "(" [[{ fp ","  }]  fp ["..."]] ")" # formal parameter list
        fp ::=  [ id ":" ] type         # formal parameter

exp ::=  id | num_lit | str_lit         # atomic expressions
        |  "(" exp ")"         # parenthesized expression
        |  exp op exp | op exp | exp "..." # in-fix, prefix, and postfix expression
exp ::= exp "(" ")"         # lambda application - zero args
        | exp exp         # lambda application - one arg
        | exp "(" exp { "," exp } ")" # lambda application - multiple args
        |  quantifier [{ exp }] type # quantification
        |  exp "." id  |  exp "." int_lit # dot qualified reference
        |  exp "." "(" exp ")"         # dot quantified reference
        |  exp "." "[" exp "]"         # indexed reference
        |  "[" [[{ exp "," }] exp ] "]" # aggregate constructor
        | "for" [ id ":" ] exp "do" exp # quantified condition
        | "?"                 # unspecified value
pronoun ::= "it" | "this" | "self" | "sim" | "system"   # pronouns
op ::= "<<"|">>"|"&" | "|" | "!" | "isa" | "~" # type operators
        |"<" | "=" | ">" | "<=" | "!=" | ">="|".." # relational operators
        | "+" | "-" | "*" | "/" | "^" # computational operators
        | "~" | "$" | "%" | "\"         # unassigned operators
        |  "can"         # special syntax operations
quantifier ::= "any" | "some" | "the" | "all" | "each" | "every"| num_lit

stat ::= "?"                 # unspecified action
        |  "{" [{ prop ";" }] "}"         # compound  statement
        |  exp [ "(" ")" ]         # procedure call - no args
        |  exp exp         # procedure call - one arg
        |  exp "(" exp { "," exp } ")" # procedure call - multiple args
        |  exp ":=" exp         # assignment statement
        | "if" exp "then" stat  [else_part] # conditional statement
        | "for" [ id ":" ] exp "do" stat  [else_part]   # iterative statement
        | "select" exp of [{ "case" type "then" stat }] [else_part]   # select statement
        | "take" exp "to" stat         # duration specification
else_part ::= ";" "else" stat
```

Appendix C. Parse Grammar

Syntax

```
1 0 ;    ,                                    middle key words
-- 0 (   [   {   if   for   select             left closed key words
1 -- )   ]   }                                 right closed key words

3 2 :   :=   is   can   take   ::=           open open key words
          then   else   do                      middle open key words
3 -- ...                                       open closed key word
-- 2 <exp>                                     expression as op, closed open key word

3 4 &   |   !   ~                             type operations
5 4 <   =   >   <=  !=  >=                    relational operations
      <<   >>   isa   has ..   ==
5 6 +   -   $   @                             additive operations
7 8 *   /   \   %                             multiplicative operations
9 8 ^                                                          exponentiation
-- 10 any   some   the   all                    quantifier
11 12 .   [   (   '   `                         reference operations


[ ]   with 1 arg is subscript, otherwise is aggregate expression
( )   with 1 arg is identity expression, otherwise is function form parameters
?     is an expression
      numeric literals not followed by an operator are quantifiers
      all other literals are expressions
                        all comments are ignored by parse (Note: maybe convert to annotation)
                        all fill including <sp>, <cr> and <tab> are ignored by parse
```

Only the following key following key word forms are syntactically legal:

Syntax

```
        "("   exp   ")"                                   # parenthesized expression
        "["   [ exp  { ","   exp } ]   "]"                # aggregate
        "{"   [ exp   [{ ";"   exp }]  ]   "}"            # scope & sequencing
        "if"   exp   "then"   exp   [ "else"   exp ]      # if conditional
        "select"   exp   { "case"   exp   "then"   exp }   [ "else"   exp ] # select conditional
        "for"   exp   "do"   exp   [ "else"   exp ]       # iteration
        exp   ":"   exp   ":="   exp                      # attribute specification
        exp   ":"   exp   [ "is"   exp ]                  # definition, fp spec
        exp   "is"   exp                                  # abbreviated definition
        exp   "can"   exp                                 # neighbor specification
        exp   "take"   exp                               # duration specification
        exp   "..."                                       # variable number of para's
        exp   exp                                         # lambda application
        exp   "("   [ exp   { ","   exp }]   ")"          # lambda application
        exp   "["   exp   "]"                             # subscription
```

Meta grammar symbols used above:

Syntax

```
        [   ] # optional
        {   } # zero or more repetitions
        "   " # encloses terminals of grammar
```

Appendix D. Metagrammar

The following meta grammar is used to define both the lexical grammar and parse grammar.

Syntax

```
abc                 any alphabetic sequence name a lexical category
"xyz"         xyz is a terminal of the language being defined
x ::= y         y is a legal expansion of category x
x y                 x juxtaposed with y in that order
x | y         x and y are alternative choices
[ x ]         x is optional
{ x }         x may be repeated one or more times
[{ x }]         x may be repeated zero or more times
x1 . . .  xn short hand for x1|x2|x3|x4|... xn
```

Appendix E. Predefined Colors

E.1. Web Colors

Easel predefines all but one of the standard Web color names. The exception is "tan", which conflicts with the tangent operator.

E.1.1. aliceblue: pattern is rgb(F0#16, F8#16, FF#16);

E.1.2. antiquewhite: pattern is rgb(FA#16, EB#16, D7#16);

E.1.3. aqua: pattern is rgb(00#16, FF#16, FF#16);

E.1.4. aquamarine: pattern is rgb(7F#16, FF#16, D4#16);

E.1.5. azure: pattern is rgb(F0#16, FF#16, FF#16);

E.1.6. beige: pattern is rgb(F5#16, F5#16, DC#16);

E.1.7. bisque: pattern is rgb(FF#16, E4#16, C4#16);

E.1.8. black: pattern is rgb(00#16, 00#16, 00#16);

E.1.9. blanchedalmond: pattern is rgb(FF#16, EB#16, CD#16);

E.1.10. blue: pattern is rgb(00#16, 00#16, FF#16);

E.1.11. blueviolet: pattern is rgb(8A#16, 2B#16, E2#16);

E.1.12. brown: pattern is rgb(A5#16, 2A#16, 2A#16);

E.1.13. burlywood: pattern is rgb(DE#16, B8#16, 87#16);

E.1.14. cadetblue: pattern is rgb(5F#16, 9E#16, A0#16);

E.1.15. chartreuse: pattern is rgb(7F#16, FF#16, 00#16);

E.1.16. chocolate: pattern is rgb(D2#16, 69#16, 1E#16);

E.1.17. coral: pattern is rgb(FF#16, 7F#16, 50#16);

E.1.18. cornflowerblue: pattern is rgb(64#16, 95#16, ED#16);

E.1.19. cornsilk: pattern is rgb(FF#16, F8#16, DC#16);

E.1.20. crimson: pattern is rgb(DC#16, 14#16, 3C#16);

E.1.21. cyan: pattern is rgb(00#16, FF#16, FF#16);

E.1.22. darkblue: pattern is rgb(00#16, 00#16, 8B#16);

E.1.23. darkcyan: pattern is rgb(00#16, 8B#16, 8B#16);

E.1.24. darkgoldenrod: pattern is rgb(B8#16, 86#16, 0B#16);

E.1.25. darkgray: pattern is rgb(A9#16, A9#16, A9#16);

E.1.26. darkgreen: pattern is rgb(00#16, 64#16, 00#16);

E.1.27. darkgrey: pattern is rgb(A9#16, A9#16, A9#16);

E.1.28. darkkhaki: pattern is rgb(BD#16, B7#16, 6B#16);

E.1.29. darkmagenta: pattern is rgb(8B#16, 00#16, 8B#16);

E.1.30. darkolivegreen: pattern is rgb(55#16, 6B#16, 2F#16);

E.1.31. darkorange: pattern is rgb(FF#16, 8C#16, 00#16);

E.1.32. darkorchid: pattern is rgb(99#16, 32#16, CC#16);

E.1.33. darkred: pattern is rgb(8B#16, 00#16, 00#16);

E.1.34. darksalmon: pattern is rgb(E9#16, 96#16, 7A#16);

E.1.35. darkseagreen: pattern is rgb(8F#16, BC#16, 8F#16);

E.1.36. darkslateblue: pattern is rgb(48#16, 3D#16, 8B#16);

E.1.37. darkslategray: pattern is rgb(2F#16, 4F#16, 4F#16);

E.1.38. darkslategrey: pattern is rgb(2F#16, 4F#16, 4F#16);

E.1.39. darkturquoise: pattern is rgb(00#16, CE#16, D1#16);

E.1.40. darkviolet: pattern is rgb(94#16, 00#16, D3#16);

E.1.41. deeppink: pattern is rgb(FF#16, 14#16, 93#16);

E.1.42. deepskyblue: pattern is rgb(00#16, BF#16, FF#16);

E.1.43. dimgray: pattern is rgb(69#16, 69#16, 69#16);

E.1.44. dimgrey: pattern is rgb(69#16, 69#16, 69#16);

E.1.45. dodgerblue: pattern is rgb(1E#16, 90#16, FF#16);

E.1.46. firebrick: pattern is rgb(B2#16, 22#16, 22#16);

E.1.47. floralwhite: pattern is rgb(FF#16, FA#16, F0#16);

E.1.48. forestgreen: pattern is rgb(22#16, 8B#16, 22#16);

E.1.49. fuchsia: pattern is rgb(FF#16, 00#16, FF#16);

E.1.50. gainsboro: pattern is rgb(DC#16, DC#16, DC#16);

E.1.51. ghostwhite: pattern is rgb(F8#16, F8#16, FF#16);

E.1.52. gold: pattern is rgb(FF#16, D7#16, 00#16);

E.1.53. goldenrod: pattern is rgb(DA#16, A5#16, 20#16);

E.1.54. gray: pattern is rgb(80#16, 80#16, 80#16);

E.1.55. grey: pattern is rgb(80#16, 80#16, 80#16);

E.1.56. green: pattern is rgb(00#16, 80#16, 00#16);

E.1.57. greenyellow: pattern is rgb(AD#16, FF#16, 2F#16);

E.1.58. honeydew: pattern is rgb(F0#16, FF#16, F0#16);

E.1.59. hotpink: pattern is rgb(FF#16, 69#16, B4#16);

E.1.60. indianred: pattern is rgb(CD#16, 5C#16, 5C#16);

E.1.61. indigo: pattern is rgb(4B#16, 00#16, 82#16);

E.1.62. ivory: pattern is rgb(FF#16, FF#16, F0#16);

E.1.63. khaki: pattern is rgb(F0#16, E6#16, 8C#16);

E.1.64. lavender: pattern is rgb(E6#16, E6#16, FA#16);

E.1.65. lavenderblush: pattern is rgb(FF#16, F0#16, F5#16);

E.1.66. lawngreen: pattern is rgb(7C#16, FC#16, 00#16);

E.1.67. lemonchiffon: pattern is rgb(FF#16, FA#16, CD#16);

E.1.68. lightblue: pattern is rgb(AD#16, D8#16, E6#16);

E.1.69. lightcoral: pattern is rgb(F0#16, 80#16, 80#16);

E.1.70. lightcyan: pattern is rgb(E0#16, FF#16, FF#16);

E.1.71. lightgoldenrodyellow: pattern is rgb(FA#16, FA#16, D2#16);

E.1.72. lightgray: pattern is rgb(D3#16, D3#16, D3#16);

E.1.73. lightgreen: pattern is rgb(90#16, EE#16, 90#16);

E.1.74. lightgrey: pattern is rgb(D3#16, D3#16, D3#16);

E.1.75. lightpink: pattern is rgb(FF#16, B6#16, C1#16);

E.1.76. lightsalmon: pattern is rgb(FF#16, A0#16, 7A#16);

E.1.77. lightseagreen: pattern is rgb(20#16, B2#16, AA#16);

E.1.78. lightskyblue: pattern is rgb(87#16, CE#16, FA#16);

E.1.79. lightslategray: pattern is rgb(77#16, 88#16, 99#16);

E.1.80. lightslategrey: pattern is rgb(77#16, 88#16, 99#16);

E.1.81. lightsteelblue: pattern is rgb(B0#16, C4#16, DE#16);

E.1.82. lightyellow: pattern is rgb(FF#16, FF#16, E0#16);

E.1.83. lime: pattern is rgb(00#16, FF#16, 00#16);

E.1.84. limegreen: pattern is rgb(32#16, CD#16, 32#16);

E.1.85. linen: pattern is rgb(FA#16, F0#16, E6#16);

E.1.86. magenta: pattern is rgb(FF#16, 00#16, FF#16);

E.1.87. maroon: pattern is rgb(80#16, 00#16, 00#16);

E.1.88. mediumaquamarine: pattern is rgb(66#16, CD#16, AA#16);

E.1.89. mediumblue: pattern is rgb(00#16, 00#16, CD#16);

E.1.90. mediumorchid: pattern is rgb(BA#16, 55#16, D3#16);

E.1.91. mediumpurple: pattern is rgb(93#16, 70#16, DB#16);

E.1.92. mediumseagreen: pattern is rgb(3C#16, B3#16, 71#16);

E.1.93. mediumslateblue: pattern is rgb(7B#16, 68#16, EE#16);

E.1.94. mediumspringgreen: pattern is rgb(00#16, FA#16, 9A#16);

E.1.95. mediumturquoise: pattern is rgb(48#16, D1#16, CC#16);

E.1.96. mediumvioletred: pattern is rgb(C7#16, 15#16, 85#16);

E.1.97. midnightblue: pattern is rgb(19#16, 19#16, 70#16);

E.1.98. mintcream: pattern is rgb(F5#16, FF#16, FA#16);

E.1.99. mistyrose: pattern is rgb(FF#16, E4#16, E1#16);

E.1.100. moccasin: pattern is rgb(FF#16, E4#16, B5#16);

E.1.101. navajowhite: pattern is rgb(FF#16, DE#16, AD#16);

E.1.102. navy: pattern is rgb(00#16, 00#16, 80#16);

E.1.103. oldlace: pattern is rgb(FD#16, F5#16, E6#16);

E.1.104. olive: pattern is rgb(80#16, 80#16, 00#16);

E.1.105. olivedrab: pattern is rgb(6B#16, 8E#16, 23#16);

E.1.106. orange: pattern is rgb(FF#16, A5#16, 00#16);

E.1.107. orangered: pattern is rgb(FF#16, 45#16, 00#16);

E.1.108. orchid: pattern is rgb(DA#16, 70#16, D6#16);

E.1.109. palegoldenrod: pattern is rgb(EE#16, E8#16, AA#16);

E.1.110. palegreen: pattern is rgb(98#16, FB#16, 98#16);

E.1.111. paleturquoise: pattern is rgb(AF#16, EE#16, EE#16);

E.1.112. palevioletred: pattern is rgb(DB#16, 70#16, 93#16);

E.1.113. papayawhip: pattern is rgb(FF#16, EF#16, D5#16);

E.1.114. peachpuff: pattern is rgb(FF#16, DA#16, B9#16);

E.1.115. peru: pattern is rgb(CD#16, 85#16, 3F#16);

E.1.116. pink: pattern is rgb(FF#16, C0#16, CB#16);

E.1.117. plum: pattern is rgb(DD#16, A0#16, DD#16);

E.1.118. powderblue: pattern is rgb(B0#16, E0#16, E6#16);

E.1.119. purple: pattern is rgb(80#16, 00#16, 80#16);

E.1.120. red: pattern is rgb(FF#16, 00#16, 00#16);

E.1.121. rosybrown: pattern is rgb(BC#16, 8F#16, 8F#16);

E.1.122. royalblue: pattern is rgb(41#16, 69#16, E1#16);

E.1.123. saddlebrown: pattern is rgb(8B#16, 45#16, 13#16);

E.1.124. salmon: pattern is rgb(FA#16, 80#16, 72#16);

E.1.125. sandybrown: pattern is rgb(F4#16, A4#16, 60#16);

E.1.126. seagreen: pattern is rgb(2E#16, 8B#16, 57#16);

E.1.127. seashell: pattern is rgb(FF#16, F5#16, EE#16);

E.1.128. sienna: pattern is rgb(A0#16, 52#16, 2D#16);

E.1.129. silver: pattern is rgb(C0#16, C0#16, C0#16);

E.1.130. skyblue: pattern is rgb(87#16, CE#16, EB#16);

E.1.131. slateblue: pattern is rgb(6A#16, 5A#16, CD#16);

E.1.132. slategray: pattern is rgb(70#16, 80#16, 90#16);

E.1.133. slategrey: pattern is rgb(70#16, 80#16, 90#16);

E.1.134. snow: pattern is rgb(FF#16, FA#16, FA#16);

E.1.135. springgreen: pattern is rgb(00#16, FF#16, 7F#16);

E.1.136. steelblue: pattern is rgb(46#16, 82#16, B4#16);

E.1.137. teal: pattern is rgb(00#16, 80#16, 80#16);

E.1.138. thistle: pattern is rgb(D8#16, BF#16, D8#16);

E.1.139. tomato: pattern is rgb(FF#16, 63#16, 47#16);

E.1.140. turquoise: pattern is rgb(40#16, E0#16, D0#16);

E.1.141. violet: pattern is rgb(EE#16, 82#16, EE#16);

E.1.142. wheat: pattern is rgb(F5#16, DE#16, B3#16);

E.1.143. white: pattern is rgb(FF#16, FF#16, FF#16);

E.1.144. whitesmoke: pattern is rgb(F5#16, F5#16, F5#16);

E.1.145. yellow: pattern is rgb(FF#16, FF#16, 00#16);

E.1.146. yellowgreen: pattern is rgb(9A#16, CD#16, 32#16);

E.2. Easel Colors

The color **Tan** is predefined as a substitute for **tan** (see above). The color **dark_gray** (or **dark_grey**) is defined to compensate for the anomaly that the standard Web color **darkgray** is actually *lighter* than gray.

E.2.1. Tan: pattern is rgb(D2#16, B4#16, 8C#16);

E.2.2. dark_gray: pattern is rgb(60#16, 60#16, 60#16);

E.2.3. dark_grey: pattern is rgb(60#16, 60#16, 60#16);

INDEX

Table of Contents