



HAL
open science

**ActorScript™ extension of C#®, Java®, Objective C®,
JavaScript®, and SystemVerilog using iAdaptive™
concurrency for antiCloud™ privacy and security**

Carl Hewitt

► **To cite this version:**

Carl Hewitt. ActorScript™ extension of C#®, Java®, Objective C®, JavaScript®, and SystemVerilog using iAdaptive™ concurrency for antiCloud™ privacy and security. Inconsistency Robustness, 2015, 978-1-84890-159-9. hal-01147821v1

HAL Id: hal-01147821

<https://hal.science/hal-01147821v1>

Submitted on 4 May 2015 (v1), last revised 1 Jan 2017 (v6)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

ActorScript™ extension of C#®, Java®, Objective C®, JavaScript®, and SystemVerilog using iAdaptive™ concurrency for antiCloud™ privacy and security

Carl Hewitt

*This article is dedicated to Alonzo Church, John McCarthy,
Ole-Johan Dahl and Kristen Nygaard.*

Message passing using types is the foundation of system communication:

- Messages are the unit of communication
- Types enable secure communication with Actors

ActorScript™ is a general purpose programming language for implementing iAdaptive™ concurrency that manages resources and demand. It is differentiated from previous languages by the following:

- Universality
 - Ability to directly specify exactly what Actors can and cannot do
 - Everything is accomplished with message passing using types including the very definition of ActorScript itself.
 - Messages can be directly communicated without requiring indirection through brokers, channels, class hierarchies, mailboxes, pipes, ports, queues *etc.* Programs do not expose low-level implementation mechanisms such as threads, tasks, locks, cores, *etc.* Application binary interfaces are afforded so that no program symbol need be looked up at runtime. Functional, Imperative, Logic, and Concurrent programs are integrated.
 - A type in ActorScript is an interface that does not name its implementations (contra to object-oriented programming languages beginning with Simula that name implementations called “classes” that are types). ActorScript can send a message to any Actor for which it has an (imported) type.
 - Concurrency can be dynamically adapted to resources available and current load.

- Safety, security and readability
 - Programs are *extension invariant*, i.e., extending a program does not change the meaning of the program that is extended.
 - Applications cannot directly harm each other.
 - Variable races are eliminated while allowing flexible concurrency.
 - Lexical singleness of purpose. Each syntactic token is used for exactly one purpose.
- Performanceⁱ
 - Imposes no overhead on implementation of Actor systems in the sense that ActorScript programs are as efficient as the same implementation in machine code. For example, message passing has essentially same overhead as procedure calls and looping.
 - Execution dynamically adjusted for system load and capacity (e.g. cores)
 - Locality because execution is not bound by a sequential global memory model
 - Inherent concurrency because execution is not limited by being restricted to communicating *sequential* processes
 - Minimize latency along critical paths

ActorScript attempts to achieve the highest level of performance, scalability, and expressibility with a minimum of primitives.

C# is a registered trademark of Microsoft, Inc.

Java and JavaScript are registered trademarks of Oracle, Inc.

Objective C is a registered trademark of Apple, Inc.

Computer software should not only work; it should also appear to work.¹

Introduction

ActorScript is based on the Actor mathematical model of computation that treats “Actors” as the universal conceptual primitive of digital computation [Hewitt, Bishop, and Steiger 1973; Hewitt 1977; Hewitt 2010a]. Actors have been used as a framework for a theoretical understanding of concurrency, and

ⁱ Performance can be tricky as illustrated by the following:

- “Those who would forever give up correctness for a little temporary performance deserve neither correctness nor performance.” [Philips 2013]
- “The key to performance is elegance, not battalions of special cases” [Jon Bentley and Doug McIlroy]
- “If you want to achieve performance, start with comprehensible.” [Philips 2013]
- Those who would forever give up performance for a feature that slows everything down deserve neither the feature nor performance.

as the theoretical basis for several practical implementations of concurrent systems.

ActorScript

ActorScript is a general purpose programming language for implementing massive local and nonlocal concurrency.

This paper makes use of the following typographical conventions that arise from underlying namespaces for types, messages, language constructs, syntax categories, *etc.*¹

- type identifiers (*e.g.*, **Integer**)
- program variables (*e.g.*, **aBalance**)
- message names (*e.g.*, **getBalance**)
- reserved words² for language constructs (*e.g.*, **Actor**)
- structures (*e.g.*, **[** and **]**)
- argument keyword (*e.g.*, **to**)
- logical variables (*e.g.*, *x*)
- comments in programs (*e.g.* `/* this is a comment */`)

There is a diagram of the syntax categories of ActorScript in an appendix of this paper in addition to an appendix with an index of symbols and names along with an explanation of the notation used to express the syntax of ActorScript.³

Actors

ActorScript is based on the Actor Model of Computation [Hewitt, Bishop, and Steiger 1973; Hewitt 2010a] in which all computational entities are Actors and all interaction is accomplished using message passing.

The Actor model is a mathematical theory that treats “*Actors*” as the universal conceptual primitive of digital computation. The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. Unlike previous models of computation, the Actor model was inspired by physical laws. The advent of massive concurrency through client-cloud computing and many-core computer architectures has galvanized interest in the Actor model.

¹ The choice of typography in terms of font and color has no semantic significance. The typography in this paper was chosen for pedagogical motivations and is in no way fundamental. Also, only the abstract syntax of ActorScript is fundamental as opposed to the surface syntax with its many symbols, *e.g.*, **→**, *etc.*

An Actor is a computational entity that, in response to a message it receives, can concurrently:

- send messages to addresses of Actors that it has
- create new Actors
- for an exclusive Actor, designate how to handle the next message it receives.

There is no assumed order to the above actions and they could be carried out concurrently. In addition two messages sent concurrently can be received in either order. Decoupling the sender from communication it sends was a fundamental advance of the Actor model enabling asynchronous communication and control structures as patterns of passing messages.

The Actor model can be used as a framework for modeling, understanding, and reasoning about, a wide range of concurrent systems. For example:

- Electronic mail (e-mail) can be modeled as an Actor system. Mail accounts are modeled as Actors and email addresses as Actor addresses.
- Web Services can be modeled with endpoints modeled as Actor addresses.
- Object-oriented programming objects with locks (e.g. as in Java and C#) can be modeled as Actors.

Actor technology will see significant application for coordinating all kinds of digital information for individuals, groups, and organizations so their information usefully links together. Information coordination needs to make use of the following information system principles:

- **Persistence.** *Information is collected and indexed.*
- **Concurrency:** *Work proceeds interactively and concurrently, overlapping in time.*
- **Quasi-commutativity:** *Information can be used regardless of whether it initiates new work or becomes relevant to ongoing work.*
- **Sponsorship:** *Sponsors provide resources for computation, i.e., processing, storage, and communications.*
- **Pluralism:** *Information is heterogeneous, overlapping and often inconsistent. There is no central arbiter of truth.*
- **Provenance:** *The provenance of information is carefully tracked and recorded.*

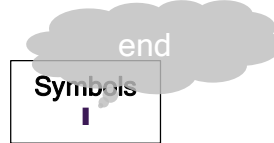
The Actor Model is designed to provide a foundation for inconsistency robust information coordination.

Syntax

To ease interoperability, ActorScript uses an intersection of the orthographic conventions of Java, JavaScript, and C++ for wordsⁱ and numbers.

Expressions

ActorScript makes use of a great many symbols to improve readability and remove ambiguity. For example the symbol “**␣**” is used as the top level terminator to designate the end of input in a read-eval-print loop. An Integrated Development Environment (IDE) can provide a table of these symbols for ease of input as explained below:ⁱⁱ



Expressions evaluate to Actors. For example, $1+3$ **␣**ⁱⁱⁱ is equivalent^{iv} to 4 **␣**.

Parentheses “(” and “)” can be used for precedence. For example using the usual precedence for operators, $3*(4+2)$ **␣** is equivalent to 18 **␣**, while $3*4+2$ **␣** is equivalent to 14 **␣**,

Identifiers, e.g., x , are expressions that can be used in other expressions. For example if x is 1 then $x+3$ **␣** is equivalent to 4 **␣**. The formal syntax of identifiers is in the following end note: **4**.

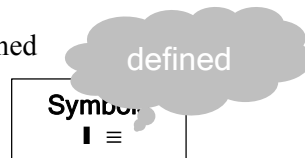
Types

Types are Actors. In this paper, Types are shown in green, e.g., **Integer**.

The formal syntax for types is in the following end note: **5**.

Definitions, i.e., \equiv

A simple definition has the name to be defined followed by “ \equiv ” followed by the definition. For example, x :**Integer** $\equiv 3$ **␣** defines the identifier x to be of type **Integer** with value 3 .



The formal syntax of a definition is in the end note: **6**.

ⁱ sometimes called "names"

ⁱⁱ Furthermore, all special symbols have ASCII equivalents for input with a keyboard. An IDE can convert ASCII for a symbol equivalent into the symbol. See table in an appendix to this article.

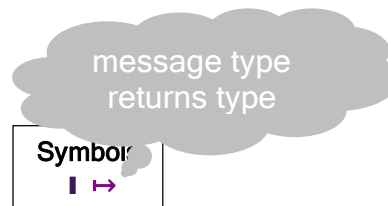
ⁱⁱⁱ An IDE can provide a box with symbols for easy input in program development. The grey callout bubble is a hover tip that appears when the cursor hovers above a symbol to explain its use.

^{iv} in the sense of having the same value and the same effects

Interfaces for procedures, *i.e.*, **Interface with []→** .

A procedure interface is used to specify the types of messages that a procedure Actor can receive. The syntax is “**Interface**” followed by an interface identifier, “**with**”, and procedure signatures in parentheses separated by commas. A procedure signature consists of a message signature with argument types delimited by “[” and “]”, followed by “→”, and a return type.ⁱ An alternative syntax (which is more like Java) is that a procedure signature can be written as a return type followed by “←”, and message signature with argument types delimited by “[” and “]”.

For example, the interfaceⁱⁱ for the overloaded⁷ procedure type **IntegerToIntegerAndVectorToVector** that takes an **Integer** argument to return an **Integer** value and a **Vector** argument and to return a **Vector** can be constructed as follows:⁸



```
Interface IntegerToIntegerAndVectorToVector with  
    [Integer]→ Integer,  
    [Vector]→ Vector. I
```

For security reasons, the type **IntegerToIntegerAndVectorToVector** is *different* from the type constructed below:ⁱⁱⁱ

```
Interface VectorToVectorAndIntegerToInteger with  
    [Vector]→ Vector,  
    [Integer]→ Integer. I
```

The formal syntax of a procedure interface is in the following end note: **9**.

ⁱ Since communicating using messages is crucial for Actor systems, messages are shown in magenta in this article. The choice of color has no semantic significance.

ⁱⁱ Every interface is a type.

ⁱⁱⁱ Merely, having procedures with the same signatures does not make **IntegerToIntegerAndVectorToVector** the same type as **VectorToVectorAndIntegerToInteger**. Types can be used to enforce security.

Procedures, *i.e.*, Actor implements []→, ¶ and §

A procedure has message formal parameters delimited by “[” and “]” followed by “→” and then the expression to be computed.ⁱ For example, [n:Integer]→n+n¶ is a (unnamed) procedure that given a message with an integer number, n, returns the number plus itself.

Procedures can be overloaded using “Actor implements”, followed by a type, followed by “using”, followed by a list of procedures separated by “¶” and terminated by “§”.ⁱⁱ For example, in the following Double is defined to implement IntegerToIntegerAndVectorToVector.

```
Double ≡ Actor implements IntegerToIntegerAndVectorToVector using
  [n:Integer]→ n+n¶
  // integer addition
  [v:Vector]→ v+v §¶
  // vector addition
```



The formal syntax of procedures is in the end note: 10.

Sending messages to procedures, *i.e.*, .[]

Sending a message to a procedure (*i.e.* “calling” a procedure with arguments) is expressed by an expression that evaluates to a procedure followed by “.”ⁱⁱⁱ followed by a message with parameter expressions delimited by “[” and “]”. For example, Square.[2+1]¶ means send Squareⁱⁱⁱ the message [3]. Thus Square.[2+1]¶ is equivalent to 9¶.

The formal syntactic definition of procedural message sending is in the end note: 12.

ⁱ Note the following crucial differences (recalling that font, color, and capitalization are of no semantic significance for identifiers although words with different capitalization are different identifiers):

- [Integer]→Integer is a procedure signature type and *not* a procedure. It is a procedure type for a procedure that takes an Integer argument and returns an Integer.
- [Integer]→Integer is a procedure and *not* a type. It is the “identity” procedure of one argument that always returns the argument.

ⁱⁱ Since both procedures and implementations can be quite large, an IDE can use these special symbols to provide additional help.

ⁱⁱⁱ As a convenience, the procedure Square can be defined to implement the type [Integer]→Integer as follows: Square.[x:Integer]:Integer ≡ x*x¶

Patterns

Patterns are fundamental to ActorScript. For example,

- 3 is a pattern that matches 3
- “abc” is a pattern that matches “abc”.
- `_` is a pattern that matches anythingⁱ
- `$$x` is a pattern that matches the value of `x`.
- `$(x+2)` is a pattern that matches the value of the expression `x+2`.
- `< 5` is a pattern that matches an integer less than 5
- `x suchThat Factorial.[x]>120` is a pattern that matches an integer whose factorial is greater than 120

Identifiersⁱⁱ can be bound using patterns as in the following examples:

- `x` is a pattern that matches “abc” and binds `x` to “abc”

Cases, *i.e.*, `⋄` ; ; `?`

Cases are used to perform conditional testing. In a Cases Expression, an expression for the value on which to perform case analysis is specified first followed by “`⋄`”ⁱⁱⁱ and then followed by a number of cases separated by “`⊖`” terminated by “`?`”¹³. A case consists of

- a pattern followed by “`⋄`” and an expression to compute the value for the case. *All of the patterns before an **else** case must be disjoint; *i.e.*, it must not be possible for more than one to match.*
- optionally (at the end of the cases) *one or more* of the following cases: “**else**” followed by an optional pattern, “`⋄`”, and an expression to compute the value for the case. An **else** case applies *only* if none of the patterns in the preceding cases^{iv} match the value on which to perform case analysis.

ⁱ e.g., `_` matches 7

ⁱⁱ An identifier is a name that is used in a program to designate an Actor

ⁱⁱⁱ “`⋄`” is fancy typography for “`?`”

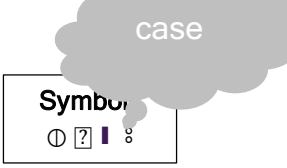
^{iv} *including* patterns in previous else cases

As an arbitrary example purely to illustrate the above, suppose that the procedure `Random` is of type `[] → Integer` in the following example:

```

Random.[ ]
0 : // Random.[ ] returned 0i
  Throwii RandomNumberException[ ]
  // throw an exception
  // because Fibonacci.[0] is undefined
1 : // Random.[ ] returned 1
  6 // the value of the cases expression is 6
else y that is < 5 :
  // Random.[ ] returned y that is not 0 or 1 and is less than 5
  Fibonacci.[y]
  // return Fibonacci of the value returned by Random.[ ]
else z :
  // Random.[ ] returned z that is not 0 or 1 and is not less than 5
  Factorial.[z]
  // return Factorial of the value returned by Random.[ ]

```



The formal syntax of cases is in the following end note: **14**.


Binding locals, *i.e.*, **Let** ← .

Local identifiers can be bound using “**Let**” followed by a pattern, “←”, an expression for the Actor to be matched followed by “,” and a list of expressions terminated with “.”. For example, `aProcedure.[“G”, “F”, “F”]` could be written as follows:

```

Let x ← “F”. // x is “F”
  aProcedure.[“G”, x, x]

```

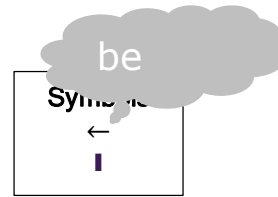


ⁱ As is standard, ActorScript uses the token “//” to begin a one-line comment.

ⁱⁱ Reserved words are shown in bold black.

Dependent bindings (in which each can depend on previous ones) can be accomplished by nesting **Let**. Also, a binding can be accomplished using a list pattern. For example:

```
Let x ← "F".           // x is "F"
  Let y ← aProcedure["G", x, x].
                // y is aProcedure["G", "F", "F"]
  anotherProcedure["x, y"]
```



The above is equivalent to

```
anotherProcedure["F", aProcedure["G", "F", "F"]]
```

The formal syntax of bindings is in the following end note: **15**.

General Message-passing interfaces

Procedure interfaces are a special case of general message-passing interfaces.

A message handler signature consists of a message name followed by argument types delimited by “[” and “]”, “→”, and a return type. For example

```
Interface Account with getBalance[] → Currency,
                        deposit[Currency] → Void,
                        withdraw[Currency] → Void. |
```

The formal syntactic definition of named-message sending is in the following end note: **16**

Actors that change, i.e., Actor and :=

Using the expressions introduced so far, actors do not change. However, some Actors change behaviors over time.

An Actor can be created using "Actor" optionally followed by the following:

- constructor name with formal arguments delimited using brackets
- declarations of variablesⁱ terminated by “.”
- implementations of interface(s).

Message handlers in an Actor execute mutually exclusively while in a region of mutual exclusion which is called “cheese.” In this paper assignable variables are colored orange, which by itself has no semantic significance, i.e., printing this article in black and white does not change any meaning. The use

ⁱ variable declarations separated by commas

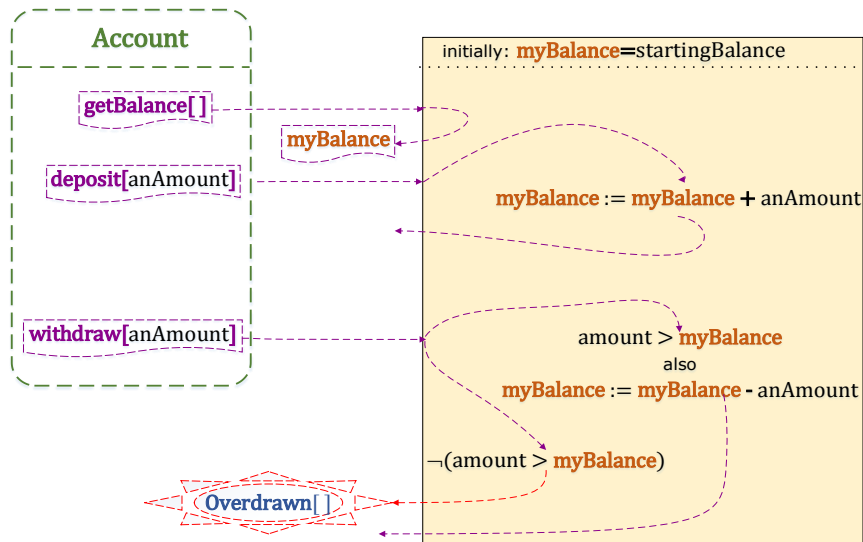
of assignments is strictly controlled in order to achieve better structured programs.¹⁷

ActorScript is referentially transparent in the sense that variable never changes while in a continuous part of the cheese.¹⁸ For example, in the **withdraw** message handler change is accomplished using the following:

Void afterward **myBalance := myBalance + anAmount**
 which returns **Void** and updates **myBalance** for the *next* message received.

Variable races are impossible in ActorScript.

Below is a diagram for an Actor, which implements **Account**:



The implementation of **Account** above can be expressed as follows:



```

Actor SimpleAccount[startingBalance:Euro]
  // SimpleAccount has [Euro]→Accounti
  myBalance := startingBalance.
  // myBalance is an assignable variable initialized with startingBalance
  implements Account using
  getBalance[ ] → myBalance¶
  deposit[anAmount] →
    Void // return Void
    afterward myBalance := myBalance+anAmount¶
      // the next message is processed with
      // myBalance reflecting the deposit
  withdraw[anAmount] →
    (amount > myBalance) ⋄
    True ⋮ Throw Overdrawn[ ]⓪
    False ⋮ Void // return Void
    afterward myBalance := myBalance-anAmount ¶§¶
      // the next message is processed with updated myBalance

```

The formal syntax of **Actor** expressions is in the following end note: 19.

Antecedents, Preparations, and Concurrency, *i.e.*, ⓪

An expression can be annotated for concurrent execution by preceding it with “⓪” indicating that the following expression should be considered for concurrent execution if resources are available. For example ⓪Factorial.[1000]+⓪Fibonacci.[2000]¶ is annotated for concurrent execution of Factorial.[1000] and Fibonacci.[2000] both of which *must* complete execution. This does not require that the executions of Factorial.[1000] and Fibonacci.[2000] actually overlap in time.

The formal syntax of explicit concurrency is in the following end note: 20.

Concurrency can be controlled using preparation that is expressed in a continuation using preparatory expressions, “●” and an expression that proceeds only *after* the preparations have been completed.

ⁱ SimpleAccount is a constructor (that can be called as a procedure) with a single argument that is of **Euro** which returns an Actor of type **Account**

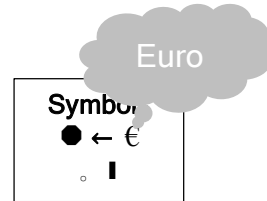
The following expression creates an account `anAccount` with initial balance €5 and then concurrently withdraws €1 and €2 in preparation for reading the balance:

```

Let anAccount ← SimpleAccount.[€6]. //€ is a reserved prefix operator
ⓂanAccount.withdraw[€1],
ⓂanAccount.withdraw[€2] ●
    // proceed only after both of the
    // withdrawals have been acknowledged
anAccount.getBalance[ ]. !

```

The above expression returns €3.



Operations are quasi-commutative to the extent that it doesn't matter in which order they occur.

Quasi-commutativity can be used to tame indeterminacy while at the same time facilitating implementations that run exponentially faster than those in the parallel lambda calculus.¹

The formal syntax of compound expressions is in the following end note: **21**

Implementing multiple interfaces , *i.e.*, `Ⓜ` and also implements

The above implementation of **Account** can be extended as follows to provide the ability to revoke some abilities to change an account.²² For example, `AccountSupervisor` below implements both the **Account** and **AccountRevoker** interfaces as an extension of the implementation `SimpleAccount`:

As illustrated below, a qualified address of an Actor can be expressed using “`Ⓜ`” followed by the name of the qualifier.²³

¹ For example, implementations using Actors of Direct Logic can be exponentially faster than implementations in the parallel lambda calculus.

```

Actor AccountSupervisor [initialBalance:Currency]
  extends SimpleAccount[initialBalance]
  withdrawableIsRevoked := False,
  depositableIsRevoked := False.
  implements AccountSupervisor using
    getRevoker[ ] → AccountRevoker
    getAccount[ ] → Account
    withdrawFee[anAmount] →
      Void afterward myBalance := myBalance - anAmount$
        // withdraw fee even if balance goes negative24
  also partially reimplements exportable Account using
    withdraw[anAmount] →
      withdrawableIsRevoked ◆
      True : Throw Revoked[ ] ⊕
      False : SimpleAccount.withdraw[anAmount] ?$
    deposit[anAmount] →
      depositableIsRevoked ◆
      True : Throw Revoked[ ] ⊕
      False : SimpleAccount.deposit[anAmount] ?$
  also implements exportable AccountRevoker using
    revokeDepositable[ ] →
      Void afterward depositableIsRevoked := True
    revokeWithdrawable[ ] →
      Void afterward withdrawableIsRevoked := True

```

For example, the following expression returns *negative* €3:

```

Let anAccountSupervisor ← AccountSupervisor.[€3].
Let anAccount ← anAccountSupervisor.getAccount[ ],
    aRevoker ← anAccountSupervisor.getRevoker[ ].
anAccount.withdraw[€2] ● // the balance is €1
aRevoker.revokeWithdrawable[ ] ●
// withdrawableIsRevoked is True
Try anAccount.withdraw[€5] // try another withdraw
  catch _ : Void ? ● // ignore the thrown exception25
// the balance remains €1
anAccountSupervisor.withdrawFee[€4] ●
// €4 is withdrawn even though withdrawableIsRevoked
anAccount.getBalance[ ] . | // the balance is negative €3

```

The formal syntax of the programs below is in the following end note: **26**

Swiss cheese

Swiss cheese [Hewitt and Atkinson 1977, 1979; Atkinson 1980]²⁷ is a generalization of mutual exclusion with the following goals:

- *Generality*: Ability to conveniently program any scheduling policy
- *Performance*: Support maximum performance in implementation, e.g., the ability to minimize locking and to avoid repeatedly recalculating a condition for proceeding.
- *Understandability*: Invariants for the variables of a mutable Actor should hold whenever entering or leaving the cheese.
- *Modularity*: Resources requiring scheduling should be encapsulated so that it is impossible to use them incorrectly.

There is a very simple Actor with the following interface that cannot be performed by a nondeterministic Turing Machine (equivalently implemented in the nondeterministic lambda calculus):

Interface Counter with `go[]` \mapsto `Void`,
`stop[]` \mapsto `Integer`. ■

An implementation of the above interface is described below.

By contrast with the nondeterministic lambda calculus, there is an always-halting Actor that when sent a `start[]` message can compute an integer of unbounded size. This is accomplished by creating a **Counter** with the following variables:

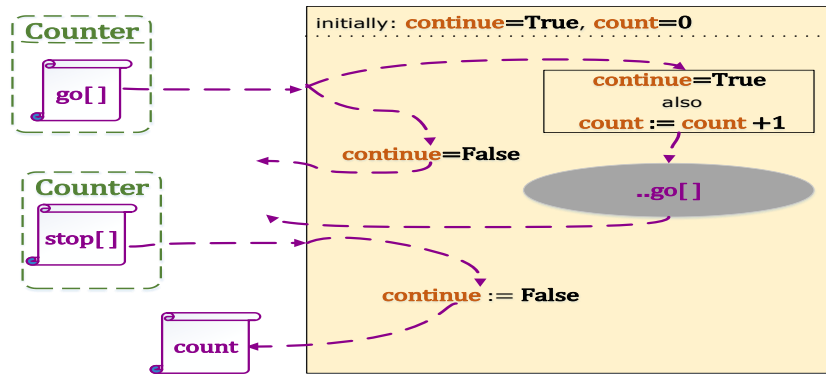
- `count` initially `0`
- `continue` initially `True`

and concurrently sending it both a `stop[]` message and a `go[]` message such that:

- When a `go[]` message is received:
 1. if `continue` is `True`, increment `count` by 1 and return the result of sending this counter a `go[]` message.
 2. if `continue` is `False`, return `Void`
- When a `stop[]` message is received, return `count` and sent `continue` to `False` for the next message received.

By the Actor Model of Computation [Clinger 1981, Hewitt 2006], the above Actor will eventually receive the `stop[]` message and return an unbounded number.

A diagram is shown below for an implementation of **Counter**. In the diagram, a hole in the cheese is highlighted in grey and variables are shown in orange. The color has no semantic significance.



ComputeUnbounded \equiv

Actor implements **Unbounded**

start[]:Integer \rightarrow // a **start** message is implemented by

Let aCounter \leftarrow SimpleCounter.**[]**. // let aCounter be a new **Counter**

ⓈaCounter.**go[]**,

// send aCounter a **go** message and *concurrently*

ⓈaCounter.**stop[]**. **|**

// return the result of sending aCounter a **stop** message

As a notational convenience, when an Actor receives message then it can send an arbitrary message to itself by prefixing it with “**..**”.

Actor SimpleCounter **[]**

count \equiv 0, // the variable **count** is initially 0

continue \equiv True.

implements **Counter** using

stop[] \rightarrow

count

// return **count**

afterward **continue** \equiv False **Ⓢ**

// **continue** is updated to **False** for the next message received

go[] \rightarrow

continue \Leftarrow

True \Leftarrow **Hole** **..go[]**

// send **go[]** to this counter after

after **count** \equiv **count**+1 **Ⓢ**

// incrementing **count**

False \Leftarrow Void **Ⓢ****|**

// if **continue** is **False**, return **Void**

Symbols

\equiv \rightarrow \Leftarrow **Ⓢ** **Ⓢ**

Ⓢ **Ⓢ** **Ⓢ** **|**

Coordinating Activities

Coordinating activities of readers and writers in a shared resource is a classic problem. The fundamental constraint is that multiple writers are not allowed to operate concurrently and a writer is not allowed operate concurrently with a reader.

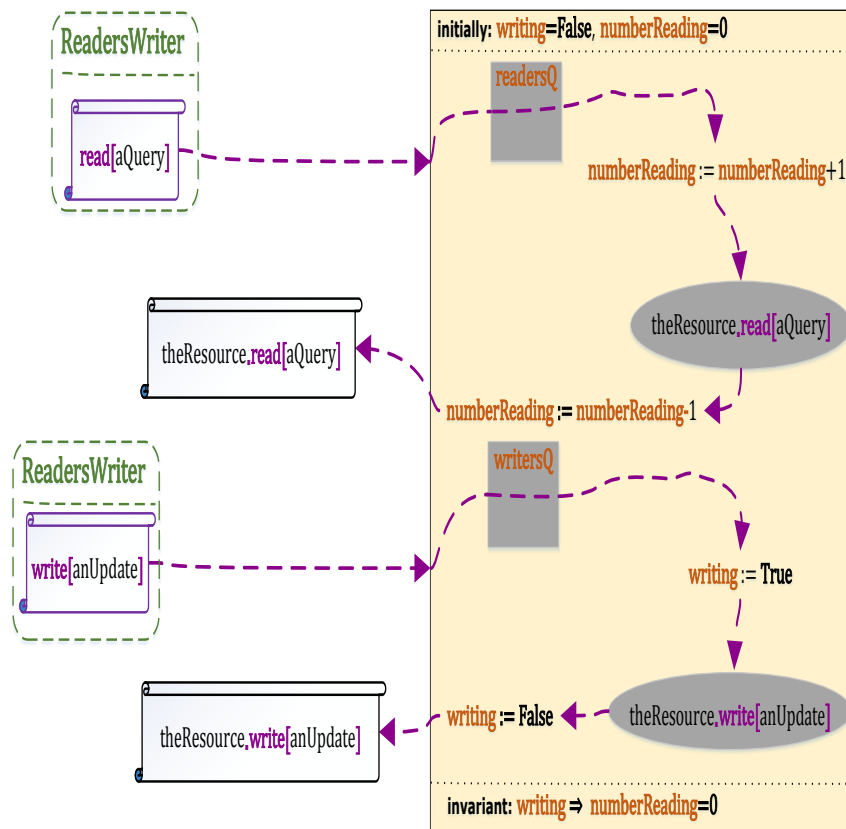
Below are two implementations of readers/writer guardians for a shared resource that implement different policies:²⁸

1. *ReadingPriority*: The policy is to permit maximum concurrency among readers without starving writers.²⁹
 - a. When no writer is waiting, all readers start as they are received.
 - b. When a writer has been received, no more readers can start.
 - c. When a writer completes, all waiting readers start even if there are writers waiting.
2. *WritingPriority*: The policy is that readers get the most recent information available without starving writers.³⁰
 - a. When no writer is waiting, all readers start as they are received.
 - b. When a writer has been received, no more readers can start.
 - c. When a writer completes, just one waiting reader is permitted to complete if there are waiting writers.

The interface for the readers/writer guardian is the same as the interface for the shared resource:

Interface ReadersWriter with `read[Query] → QueryAnswer,`
`write[Update] → Void. ▮`

Cheese diagram for **ReadersWriter** implementations:



Note:

1. At most one activity is allowed to execute in the cheese.
2. The value of a variableⁱ changes only when leaving the cheese.ⁱⁱ

The formal syntax of the programs below is in the following end note: **31**

ⁱ A variable is orange in the diagram

ⁱⁱ Of course, other external Actors can change.

Illustration of writing-priority:

Actor WritingPriority[theResource:ReadersWriter]
 invariants **writing**⇒ **numberReading**=0.
 queues **readersQ**, **writersQ**.
writing := False,
numberReading:PositiveInteger:= 0.
 implements ReadersWriter using
 read[query]→
 (**writing** ∨ ¬Empty **writersQ**) ◆
 True : **Enqueue readersQ** ● // leave cheese while in **readersQ**
 backout ¬**writing** ∧ **numberReading**=0 ∧ IsEmpty **readersQ** ◆
 True : Void **permit writersQ** ⊖
 False : Void ?
 Void ⊖
 False : Void ? ●
 Preconditions ¬**writing**.
 Hole theResource.**read**[query]
 after IsEmpty **writersQ** ◆
 True : **Permit readersQ** **always numberReading++** ⊖
 False : **Also numberReading++** ?
 afterward
 (IsEmpty **writersQ**) ◆
 True : **permit readersQ** **always numberReading--** ⊖
 False : **numberReading** ◆
 1 : **permit writersQ** **always numberReading--** ⊖
 (> 1) : **numberReading--** ? ? . †
 write[update]→
 (**numberReading**>0 ∨ ¬IsEmpty **readersQ** ∨ **writing** ∨ ¬IsEmpty **writersQ**) ◆
 True : **Enqueue writersQ** ● // leave cheese while in **writersQ**
 backout (IsEmpty **writersQ** ∧ ¬**writing**) ◆
 True : Void **permit readersQ** ⊖
 False : Void ?
 Void ⊖
 False : Void ? ●
 Preconditions **numberReading**=0, ¬**writing**.
 Hole theResource.**write**[update]
 after **writing** := True
 afterward
 (IsEmpty **readersQ**) ◆
 True : **permit writersQ** **always writing := False** ⊖
 False : **permit readersQ** **always writing := False** ? . †

Symbols	
≡	→ : ⊖ ∧ ∨ ¬
?	† § †

The formal syntax of queue management in cheese is in the following end note: 36.

Conclusion

Before long, we will have billions of chips, each with hundreds of hyper-threaded cores executing hundreds of thousands of threads. Consequently, GOFIP (Good Old-Fashioned Imperative Programming) paradigm must be fundamentally extended. ActorScript is intended to be a contribution to this extension.

Acknowledgements

Important contributions to the semantics of Actors have been made by: Gul Agha, Beppe Attardi, Henry Baker, Will Clinger, Irene Greif, Carl Manning, Ian Mason, Ugo Montanari, Maria Simi, Scott Smith, Carolyn Talcott, Prasanna Thati, and Aki Yonezawa.

Important contributions to the implementation of Actors have been made by: Bill Athas, Russ Atkinson, Beppe Attardi, Henry Baker, Gerry Barber, Peter Bishop, Nanette Boden, Jean-Pierre Briot, Bill Dally, Peter de Jong, Jessie Dedecker, Ken Kahn, Henry Lieberman, Carl Manning, Mark S. Miller, Tom Reinhardt, Chuck Seitz, Dale Schumacher, Richard Steiger, Dan Theriault, Mario Tokoro, Darrell Woelk, and Carlos Varela.

Research on the Actor model has been carried out at Caltech Computer Science, Kyoto University Tokoro Laboratory, MCC, MIT Artificial Intelligence Laboratory, SRI, Stanford University, University of Illinois at Urbana-Champaign Open Systems Laboratory, Pierre and Marie Curie University (University of Paris 6), University of Pisa, University of Tokyo Yonezawa Laboratory and elsewhere.

The members of the Silicon Valley Friday AM group made valuable suggestions for improving this paper. Discussions with Blaine Garst were helpful in the development of the implementation of Swiss cheese that doesn't hold a lock as well providing background on the historical development of interfaces. Patrick Beard found bugs and suggested improvements in presentation. Fanya S. Montalvo and Ike Nassi suggested simplifying the syntax. Dale Schumacher found many typos, suggested including a syntax diagram, and suggested improvements to the syntax of collections, binding and assignment. In particular, Dale contributed greatly to the development of the lock-free¹ implementation of cheese in the appendix. Stuart Bailey and Chip Morningstar provided an excellent critique with many useful comments and suggestions.

¹ In the sense that the implementation holds a hardware lock.

ActorScript is intended to provide a foundation for information coordination in client-cloud computing that protects citizens sensitive information [Hewitt 2009b].

Bibliography

- Hal Abelson and Gerry Sussman *Structure and Interpretation of Computer Programs* 1984.
- Paul Abrahams. *A final solution to the Dangling else of ALGOL 60 and related languages* CACM. September 1966.
- Sarita Adve and Hans-J. Boehm *Memory Models: A Case for Rethinking Parallel Languages and Hardware* CACM. August 2010.
- Mikael Amborn. *Facet-Oriented Program Design*. LiTH-IDA-EX-04/047-SE Linköpings Universitet. 2004.
- Joe Armstrong *History of Erlang* HOPL III. 2007.
- Joe Armstrong. *Erlang*. CACM. September 2010/
- William Athas and Charles Seitz *Multicomputers: message-passing concurrent computers* IEEE Computer August 1988.
- William Athas and Nanette Boden *Cantor: An Actor Programming System for Scientific Computing* in Proceedings of the NSF Workshop on Object-Based Concurrent Programming. 1988. Special Issue of SIGPLAN Notices.
- Russ Atkinson. *Automatic Verification of Serializers* MIT Doctoral Dissertation. June, 1980.
- Henry Baker. *Actor Systems for Real-Time Computation* MIT EECS Doctoral Dissertation. January 1978.
- Henry Baker and Carl Hewitt *The Incremental Garbage Collection of Processes* Proceeding of the Symposium on Artificial Intelligence Programming Languages. SIGPLAN Notices 12, August 1977.
- Paul Baran. *On Distributed Communications Networks* IEEE Transactions on Communications Systems. March 1964.
- Gerry Barber. *Reasoning about Change in Knowledgeable Office Systems* MIT EECS Doctoral Dissertation. August 1981.
- Philippe Besnard and Anthony Hunter. *Quasi-classical Logic: Non-trivializable classical reasoning from inconsistent information* Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Springer LNCS. 1995.
- Peter Bishop *Very Large Address Space Modularly Extensible Computer Systems* MIT EECS Doctoral Dissertation. June 1977.
- Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007a) *Interactive small-step algorithms I: Axiomatization* Logical Methods in Computer Science. 2007.
- Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman (2007b) *Interactive small-step algorithms II: Abstract state machines and the characterization theorem*. Logical Methods in Computer Science. 2007.
- Per Brinch Hansen *Monitors and Concurrent Pascal: A Personal History* CACM 1996.
- Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Nielsen, Satish Thatte, Dave Winer. *Simple Object Access Protocol (SOAP) 1.1* W3C Note. May 2000.
- Jean-Pierre Briot. *Acttalk: A framework for object-oriented concurrent programming-design and experience* 2nd France-Japan workshop. 1999.

- Jean-Pierre Briot. *From objects to Actors: Study of a limited symbiosis in Smalltalk-80* Rapport de Recherche 88-58, RXF-LITP. Paris, France. September 1988.
- Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. *Modula-3 report (revised)* DEC Systems Research Center Research Report 52. November 1989.
- Luca Cardelli and Andrew Gordon *Mobile Ambients* FoSSaCS'98.
- Arnaud Carayol, Daniel Hirschhoff, and Davide Sangiorgi. *On the representation of McCarthy's amb in the π -calculus* "Theoretical Computer Science" February 2005.
- Alonzo Church "A Set of postulates for the foundation of logic (1&2)" *Annals of Mathematics*. Vol. 33, 1932. Vol. 34, 1933.
- Alonzo Church *The Calculi of Lambda-Conversion* Princeton University Press. 1941.
- Will Clinger. *Foundations of Actor Semantics* MIT Mathematics Doctoral Dissertation. June 1981.
- Tyler Close *Web-key: Mashing with Permission* WWW'08.
- Eric Crahen. *Facet: A pattern for dynamic interfaces*. CSE Dept. SUNY at Buffalo. July 22, 2002.
- Haskell Curry and Robert Feys. *Combinatory Logic*. North-Holland. 1958.
- Ole-Johan Dahl and Kristen Nygaard. "Class and subclass declarations" *IFIP TC2 Conference on Simulation Programming Languages*. 1967.
- William Dally and Wills, D. *Universal mechanisms for concurrency* PARLE '89.
- William Dally, et al. *The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms* IEEE Micro. April 1992.
- Jack Dennis and Earl Van Horn. *Programming Semantics for Multiprogrammed Computations* CACM. March 1966.
- Edsger Dijkstra. *Cooperating sequential processes* Technical Report EWD-123, Technological University, Eindhoven, The Netherlands. 1965.
- Edsger Dijkstra. *Go To Statement Considered Harmful* Letter to Editor CACM. March 1968.
- Jason Eisner and Nathaniel W. Filardo. *Dyna: Extending Datalog for modern AI*. Datalog Reloaded. Springer. 2011.
- Arthur Fine. *The Shaky Game: Einstein Realism and the Quantum Theory* University of Chicago Press, Chicago, 1986.
- Frederic Fitch. *Symbolic Logic: an Introduction*. Ronald Press. 1952.
- Nissim Francez, Tony Hoare, Daniel Lehmann, and Willem-Paul de Roever. *Semantics of nondeterminism, concurrency, and communication* Journal of Computer and System Sciences. December 1979.
- Christopher Fuchs *Quantum mechanics as quantum information (and only a little more)* in A. Khrenikov (ed.) *Quantum Theory: Reconstruction of Foundations* (Vaxjo: Vaxjo University Press, 2002).
- Blaine Garst. *Origin of Interfaces* Email to Carl Hewitt on October 2, 2009.
- Elihu M. Gerson. *Prematurity and Social Worlds* in *Prematurity in Scientific Discovery*. University of California Press. 2002.
- Andreas Glausch and Wolfgang Reisig. *Distributed Abstract State Machines and Their Expressive Power* Informatik Berichete 196. Humboldt University of Berlin. January 2006.
- Brian Goetz [State of the Lambda](#) Brian Goetz's Oracle Blog. July 6, 2010.
- Adele Goldberg and Alan Kay (ed.) *Smalltalk-72 Instruction Manual* SSL 76-6. Xerox PARC. March 1976.

- Dina Goldin and Peter Wegner. *The Interactive Nature of Computing: Refuting the Strong Church-Turing Thesis* Minds and Machines March 2008.
- Cordell Green. *Application of Theorem Proving to Problem Solving* IJCAI'69.
- Irene Greif and Carl Hewitt. *Actor Semantics of PLANNER-73* Conference Record of ACM Symposium on Principles of Programming Languages. January 1975.
- Irene Greif. *Semantics of Communicating Parallel Processes* MIT EECS Doctoral Dissertation. August 1975.
- William Gropp, et. al. *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press. 1998
- Pat Hayes *Some Problems and Non-Problems in Representation Theory* AISB. Sussex. July, 1974
- Werner Heisenberg. *Physics and Beyond: Encounters and Conversations* translated by A. J. Pomerans (Harper & Row, New York, 1971), pp. 63 – 64.
- Carl Hewitt. *More Comparative Schematology* MIT AI Memo 207. August 1970.
- Carl Hewitt, Peter Bishop and Richard Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence* IJCAI'73.
- Carl Hewitt, et al. *Actor Induction and Meta-evaluation* Conference Record of ACM Symposium on Principles of Programming Languages, January 1974.
- Carl Hewitt and Henry Lieberman. *Design Issues in Parallel Architecture for Artificial Intelligence* MIT AI memo 750. Nov. 1983.
- Carl Hewitt, Tom Reinhardt, Gul Agha, and Giuseppe Attardi *Linguistic Support of Receptionists for Shared Resources* MIT AI Memo 781. Sept. 1984.
- Carl Hewitt, et al. *Behavioral Semantics of Nonrecursive Control Structure* Proceedings of *Colloque sur la Programmation*, April 1974.
- Carl Hewitt. *How to Use What You Know* IJCAI. September, 1975.
- Carl Hewitt. *Viewing Control Structures as Patterns of Passing Messages* AI Memo 410. December 1976. *Journal of Artificial Intelligence*. June 1977.
- Carl Hewitt and Henry Baker *Laws for Communicating Parallel Processes* IFIP-77, August 1977.
- Carl Hewitt and Russ Atkinson. *Specification and Proof Techniques for Serializers* IEEE Journal on Software Engineering. January 1979.
- Carl Hewitt, Beppe Attardi, and Henry Lieberman. *Delegation in Message Passing* Proceedings of First International Conference on Distributed Systems Huntsville, AL. October 1979.
- Carl Hewitt and Gul Agha. *Guarded Horn clause languages: are they deductive and Logical?* in *Artificial Intelligence at MIT*, Vol. 2. MIT Press 1991.
- Carl Hewitt and Jeff Inman. *DAI Betwixt and Between: From "Intelligent Agents" to Open Systems Science* IEEE Transactions on Systems, Man, and Cybernetics. Nov./Dec. 1991.
- Carl Hewitt and Peter de Jong. *Analyzing the Roles of Descriptions and Actions in Open Systems* Proceedings of the National Conference on Artificial Intelligence. August 1983.
- Carl Hewitt. (2006). "What is Commitment? Physical, Organizational, and Social" *COIN@AAMAS'06*. (Revised version to be published in Springer Verlag Lecture Notes in Artificial Intelligence. Edited by Javier Vázquez-Salceda and Pablo Noriega. 2007) April 2006.
- Carl Hewitt (2007a). "Organizational Computing Requires Unstratified Paraconsistency and Reflection" *COIN@AAMAS*. 2007.

- Carl Hewitt (2008a) [Norms and Commitment for iOrgs™ Information Systems: Direct Logic™ and Participatory Argument Checking](#) ArXiv 0906.2756.
- Carl Hewitt (2008b) “Large-scale Organizational Computing requires Unstratified Reflection and Strong Paraconsistency” *Coordination, Organizations, Institutions, and Norms in Agent Systems III* Jaime Sichman, Pablo Noriega, Julian Padget and Sascha Ossowski (ed.). Springer-Verlag. <http://organizational.carlhewitt.info/>
- Carl Hewitt (2008e). *ORGs for Scalable, Robust, Privacy-Friendly Client Cloud Computing* IEEE Internet Computing September/October 2008.
- Carl Hewitt (2008f) [Common sense for concurrency and inconsistency robustness using Direct Logic™ and the Actor Model](#) ArXiv 0812.4852.
- Carl Hewitt (2009a) *Perfect Disruption: The Paradigm Shift from Mental Agents to ORGs* IEEE Internet Computing. Jan/Feb 2009.
- Carl Hewitt (2009b) [A historical perspective on developing foundations for client-cloud computing: iConsult™ & iEntertain™ Apps using iInfo™ Information Integration for iOrgs™ Information Systems](#) (Revised version of “Development of Logic Programming: What went wrong, What was done about it, and What it might mean for the future” AAAI Workshop on What Went Wrong. AAAI-08.) ArXiv 0901.4934.
- Carl Hewitt (2013) *Inconsistency Robustness in Logic Programs* ArXiv 0904.3036
- Carl Hewitt (2010a) [Actor Model of Computation](#) arXiv:1008.1459
- Carl Hewitt (2010b) *iTooling™: Infrastructure for iAdaptive™ Concurrency*
- Carl Hewitt (editor). [Inconsistency Robustness 1011](#) Stanford University. 2011.
- Carl Hewitt, Erik Meijer, and Clemens Szyperski “[The Actor Model \(everything you wanted to know, but were afraid to ask\)](#)” <http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask> Microsoft Channel 9. April 9, 2012.
- Carl Hewitt. [“Health Information Systems Technologies”](#) <http://ee380.stanford.edu/cgi-bin/videologger.php?target=120606-ee380-300.aspx> Slides for this video: <http://HIST.carlhewitt.info> Stanford CS Colloquium. June 6, 2012.
- Carl Hewitt. *What is computation? Actor Model versus Turing's Model* in “A Computable Universe: Understanding Computation & Exploring Nature as Computation”. edited by Hector Zenil. World Scientific Publishing Company. 2012.
- Tony Hoare *Quick sort* Computer Journal 5 (1) 1962.
- Tony Hoare *Monitors: An Operating System Structuring Concept* CACM. October 1974.
- Tony Hoare. *Communicating sequential processes* CACM. August 1978.
- Tony Hoare. *Communicating Sequential Processes* Prentice Hall. 1985.
- Tony Hoare. *Null References: The Billion Dollar Mistake*. QCon. August 25, 2009.
- W. Horwat, Andrew Chien, and William Dally. *Experience with CST: Programming and Implementation* PLDI. 1989.
- Anthony Hunter. *Reasoning with Contradictory Information using Quasi-classical Logic* Journal of Logic and Computation. Vol. 10 No. 5. 2000.
- M. Jammer *The EPR Problem in Its Historical Development* in Symposium on the Foundations of Modern Physics: 50 years of the Einstein-Podolsky-Rosen Gedankenexperiment, edited by P. Lahti and P. Mittelstaedt. World Scientific. Singapore. 1985.

- Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne. *Concurrent Haskell*, POPL'96.
- Ken Kahn. *A Computational Theory of Animation* MIT EECS Doctoral Dissertation. August 1979.
- Alan Kay. "Personal Computing" in *Meeting on 20 Years of Computing Science* Istituto di Elaborazione della Informazione, Pisa, Italy. 1975. <http://www.mprove.de/diplom/gui/Kay75.pdf>
- Frederick Knabe *A Distributed Protocol for Channel-Based Communication with Choice* PARLE'92.
- Bill Kornfeld and Carl Hewitt. *The Scientific Community Metaphor* IEEE Transactions on Systems, Man, and Cybernetics. January 1981.
- Bill Kornfeld. *Parallelism in Problem Solving* MIT EECS Doctoral Dissertation. August 1981.
- Robert Kowalski. *A proof procedure using connection graphs* JACM. October 1975.
- Robert Kowalski *Algorithm = Logic + Control* CACM. July 1979.
- Robert Kowalski. *Response to questionnaire* Special Issue on Knowledge Representation. SIGART Newsletter. February 1980.
- Robert Kowalski (1988a) *The Early Years of Logic Programming* CACM. January 1988.
- Robert Kowalski (1988b) *Logic-based Open Systems* Representation and Reasoning. Stuttgart Conference Workshop on Discourse Representation, Dialogue tableaux and Logic Programming. 1988.
- Edya Ladan-Mozes and Nir Shavit. *An Optimistic Approach to Lock-Free FIFO Queues* Distributed Computing. Springer. 2004.
- Leslie Lamport *How to make a multiprocessor computer that correctly executes multiprocess programs* IEEE Transactions on Computers. 1979.
- Peter Landin. *A Generalization of Jumps and Labels* UNIVAC Systems Programming Research Report. August 1965. (Reprinted in *Higher Order and Symbolic Computation*. 1998)
- Peter Landin *A correspondence between ALGOL 60 and Church's lambda notation* CACM. August 1965.
- Edward Lee and Stephen Neuendorffer *Classes and Subclasses in Actor-Oriented Design*. Conference on Formal Methods and Models for Codesign (MEMOCODE). June 2004.
- Steven Levy *Hackers: Heroes of the Computer Revolution* Doubleday. 1984.
- Henry Lieberman. *An Object-Oriented Simulator for the Apiary* Conference of the American Association for Artificial Intelligence, Washington, D. C., August 1983
- Henry Lieberman. *Thinking About Lots of Things at Once without Getting Confused: Parallelism in Act 1* MIT AI memo 626. May 1981.
- Henry Lieberman. *A Preview of Act 1* MIT AI memo 625. June 1981.
- Henry Lieberman and Carl Hewitt. *A real Time Garbage Collector Based on the Lifetimes of Objects* CACM June 1983.
- Barbara Liskov and Liuba Shrira *Promises: Linguistic Support for Efficient Asynchronous Procedure Calls* SIGPLAN'88.
- Barbara Liskov and Jeannette Wing . *A behavioral notion of subtyping*, TOPLAS, November 1994.
- Carl Manning. *Traveler: the Actor observatory* ECOOP 1987. Also appears in Lecture Notes in Computer Science, vol. 276.

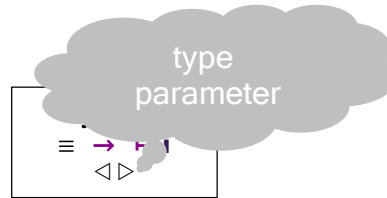
- Carl Manning. *Acore: The Design of a Core Actor Language and its Compile* Master Thesis. MIT EECS. May 1987.
- Satoshi Matsuoka and Aki Yonezawa. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages Research Directions in Concurrent Object-Oriented Programming* MIT Press. 1993.
- John McCarthy *Programs with common sense* Symposium on Mechanization of Thought Processes. National Physical Laboratory, UK. Teddington, England. 1958.
- John McCarthy. *A Basis for a Mathematical Theory of Computation* Western Joint Computer Conference. 1961.
- John McCarthy, Paul Abrahams, Daniel Edwards, Timothy Hart, and Michael Levin. *Lisp 1.5 Programmer's Manual* MIT Computation Center and Research Laboratory of Electronics. 1962.
- John McCarthy. *Situations, actions and causal laws* Technical Report Memo 2, Stanford University Artificial Intelligence Laboratory. 1963.
- John McCarthy and Patrick Hayes. *Some Philosophical Problems from the Standpoint of Artificial Intelligence* Machine Intelligence 4. Edinburgh University Press. 1969.
- Alexandre Miquel. *A strongly normalising Curry-Howard correspondence for IZF set theory* in Computer science Logic Springer. 2003
- Giuseppe Milicia and Vladimiro Sassone. *The Inheritance Anomaly: Ten Years After SAC*. Nicosia, Cyprus. March 2004.
- Mark S. Miller *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control* Doctoral Dissertation. John Hopkins. 2006.
- Mark S. Miller *et. al.* *Bringing Object-orientation to Security Programming*. YouTube. November 3, 2011.
- George Milne and Robin Milner. "Concurrent processes and their syntax" *JACM*. April, 1979.
- Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics* Chapman and Hall. 1976.
- Robin Milner. *Logic for Computable Functions: description of a machine implementation*. Stanford AI Memo 169. May 1972
- Robin Milner *Processes: A Mathematical Model of Computing Agents* Proceedings of Bristol Logic Colloquium. 1973.
- Robin Milner *Elements of interaction: Turing award lecture* CACM. January 1993.
- Marvin Minsky (ed.) *Semantic Information Processing* MIT Press. 1968.
- Eugenio Moggi *Computational lambda-calculus and monads* IEEE Symposium on Logic in Computer Science. Asilomar, California, June 1989.
- Allen Newell and Herbert Simon. *The Logic Theory Machine: A Complex Information Processing System*. Rand Technical Report P-868. June 15, 1956
- Carl Petri. *Kommunikation mit Automate* Ph. D. Thesis. University of Bonn. 1962.
- Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. *A semantics for imprecise exceptions* Conference on Programming Language Design and Implementation. 1999.
- Paul Philips. *We're Doing It all Wrong* Pacific Northwest Scala 2013.
- Gordon Plotkin. *A powerdomain construction* SIAM Journal of Computing. September 1976.
- George Polya (1957) *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving Combined Edition* Wiley. 1981.

- Karl Popper (1935, 1963) *Conjectures and Refutations: The Growth of Scientific Knowledge* Routledge. 2002.
- John Reppy, Claudio Russo, and Yingqi Xiao *Parallel Concurrent ML* ICFP'09.
- John Reynolds. *Definitional interpreters for higher order programming languages* ACM Conference Proceedings. 1972.
- Bill Roscoe. *The Theory and Practice of Concurrency* Prentice-Hall. Revised 2005.
- Kenneth Ross, Yehoshua Sagiv. *Monotonic aggregation in deductive databases*. Principles of Distributed Systems. June 1992
- Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971
- Charles Seitz. *The Cosmic Cube* CACM. Jan. 1985.
- Peter Sewell, et. al. *x86-TSO: A Rigorous and Usable Programmer's Model for x86 Microprocessors* CACM. July 2010.
- Michael Smyth. *Power domains* Journal of Computer and System Sciences. 1978.
- Guy Steele, Jr. *Lambda: The Ultimate Declarative* MIT AI Memo 379. November 1976.
- Guy Steele, Jr. *Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO*. MIT AI Lab Memo 443. October 1977.
- Gunther Stent. *Prematurity and Uniqueness in Scientific Discovery* Scientific American. December, 1972.
- Bjarne Stroustrup *Programming Languages — C++ ISO N2800*. October 10, 2008.
- Gerry Sussman and Guy Steele *Scheme: An Interpreter for Extended Lambda Calculus* AI Memo 349. December, 1975.
- David Taenzer, Murthy Ganti, and Sunil Podar, *Problems in Object-Oriented Software Reuse* ECOOP'89.
- Daniel Theriault. *A Primer for the Act-1 Language* MIT AI memo 672. April 1982.
- Daniel Theriault. *Issues in the Design and Implementation of Act 2* MIT AI technical report 728. June 1983.
- Hayo Thielecke *An Introduction to Landin's "A Generalization of Jumps and Labels"* Higher-Order and Symbolic Computation. 1998.
- Dave Thomas and Brian Barry. *Using Active Objects for Structuring Service Oriented Architectures: Anthropomorphic Programming with Actors* Journal of Object Technology. July-August 2004.
- Kazunori Ueda *A Pure Meta-Interpreter for Flat GHC, A Concurrent Constraint Language* Computational Logic: Logic Programming and Beyond. Springer. 2002.
- Darrell Woelk. *Developing InfoSleuth Agents Using Rosette: An Actor Based Language* Proceedings of the CIKM '95 Workshop on Intelligent Information Agents. 1995.
- Akinori Yonezawa, Ed. *ABCL: An Object-Oriented Concurrent System* MIT Press. 1990.
- Aki Yonezawa *Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics* MIT EECS Doctoral Dissertation. December 1977.
- Hadasa Zuckerman and Joshua Lederberg. *Postmature Scientific Discovery?* Nature. December, 1986.

Appendix 1. Extreme ActorScript

Parameterized Types, *i.e.*, \langle , \rangle

Parameterized Types are specialized using other types delimited by “ \langle ” and “ \rangle ”:



```
Double<aType> ≡
  Actor implements SingleArgument<aType> using
    // SingleArgument<aType> ≡ [aType]→aType
    [x]→ x+x §! // addition for aType
```

The formal syntax of parameterized types is in the following end note: 37 .

Type Discrimination, *i.e.*, Discrimination, Δ and ∇

A discrimination is a type of alternatives differentiated by type using “**Discrimination**” followed by a type name, “**between**”, a list of types separated using “,” terminated by “.”. A discriminate can be selected as follow:

- In an expression, by using an expression followed by “ Δ ” and the type to be selected.
- In a pattern, by using a pattern followed by Δ and the type to be selected

For example, consider the following definition:

Discrimination IntegerOrFloat between Integer, Float. !

Consequently,

- $(\text{IntegerOrFloat}\langle\text{Integer}\rangle[3])\Delta\text{Integer}$ is equivalent to 3 !
- $(\text{IntegerOrFloat}\langle\text{Float}\rangle[3.0])\Delta\text{Integer}$ throws an exception because **Integer** is not the same as the discriminant **Float**.
- The pattern $x\Delta\text{Float}$ matches $\text{IntegerOrFloat}\langle\text{Float}\rangle[3.0]$ and binds x to 3.0 .
- The expression below is equivalent to 4.0 !:
 $\text{IntegerOrFloat}\langle\text{Float}\rangle[3.0] \diamond y\Delta\text{Integer} \circ y-1 \oplus$
 $x\Delta\text{Float} \circ x+1 \text{ ?}!$

A nullable is a discrimination:

Discrimination Nullable<aType> between aType, Null<aType>. !

There is exactly one Actor that is of type $\text{Null}\langle\text{aType}\rangle$, namely $\text{Null } \text{aType}$.

A nullable can be created as follows:

Nullable x:aType ≡ Nullable<aType>[x]

Basic (whose is understood by the pattern matcher) can be defined as follows:

Discrimination Basic between Atomic, Collective. **|**

Discrimination Elemental between

Number, Character, String, Boolean, Nullable<Basic>. **|**

Discrimination Nonelemental between

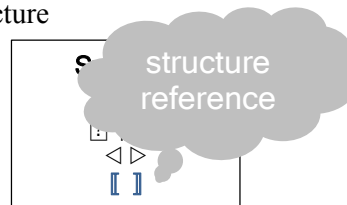
List<Basic>, Set<Basic>, Multiset<Basic>, Map<Basic, Basic>. **|**

The formal syntax of type discrimination is in the following end note: **38**.

Structures, i.e., Structure

A structure can be defined using a structure identifier followed a list of the parts enclosed by “[” and “]”.

For example, the structure **Leaf** can be defined as follows:



Structure Leaf<aType>[aType] **|** // a terminal must be of type **aType**

For example,

- The expression **Let** $x^i \leftarrow 3$. **Leaf<Integer>[x]** **|** is equivalent to **Leaf<Integer>[3]** **|**
- The pattern **Leaf<Integer>[x]** matches **Leaf<Integer>[3]** and binds x to 3.

The formal syntax of structures is in the following end note: **39**

Structures with named fields, i.e., \boxplus and $:\boxplus$

The structure **Fork** can be defined as follows:

Discrimination Tree<aType> between

Leaf<aType>, Fork<aType>. **|**

Structure Fork<aType>[left \boxplus Tree<aType>, right \boxplus Tree<aType>]

flip[]:Fork<aType> \rightarrow

Fork<aType>[left \boxplus right, right \boxplus left] **|**

// flip the branches

ⁱ x is of type **Integer**

For example,

- The expression
`Let x ← 3. Fork<Integer>[left ⊞ Leaf<Integer>[x],
right ⊞ Leaf<Integer>[x+1]]`■
is equivalent to the following:
`Fork<Integer>[left ⊞ Leaf<Integer>[3],
right ⊞ Leaf<Integer>[4]]`■
- The pattern `Fork<Integer>[left ⊞ x, right ⊞ y]` matches
`Fork<Integer>[Leaf<Integer>[6], Leaf<Integer>[6]]` and binds x
to `Leaf<Integer>[5]` and y to `Leaf<Integer>[6]`.

The formal syntax structures with named fields is in the following end note:
40.

Processing Exceptions, i.e., Try catch ◆ ⊞ , ⊞ ? and Try cleanup

It is useful to be able to catch exceptions. The following illustration returns the string “This is a test.”:

```
Try Throw Exception["This is a test." ] catch ◆  
Exception[aString] ⊞ aString ?
```

The following illustration performs `Reset.[]` and then rethrows
`Exception["This is another test."]`:

```
Try Throw Exception["This is another test." ] cleanup Reset.[ ] ■
```

The formal syntax of processing exceptions is in the following end note: **41.**

Runtime Requirements, *i.e.*, Preconditions and postcondition

A runtime requirement throws exception an exception if does not hold.

For example, the following expression throws an exception that the requirement $x \geq 0$ doesn't hold:

```
Let x ← -1.  
Preconditions  $x \geq 0$ .  
SquareRoot.[x]■
```

Post conditions can be tested using a procedure. For example, the following expression throws an exception that **postcondition** failed because square root of 2 is not less than 1:

```
SquareRoot.[2] postcondition [y:Float] →  $y < 1$ ■
```

The formal syntax requirements is in the following end note: **42**.

Polymorphism

Polymorphism provides for multiple implementations of a type. For example, **Cartesian** Actors that implement **Complex**ⁱ can be defined as follows:

```
Actor Cartesian[myReal:Float default 0, myImaginary:Float default 0]
implements Complex using // construct a Cartesian of type Complex
  realPart[ ]→ myReal¶
  imaginaryPart[ ]→ myImaginary¶
  magnitude[ ]→
    SquareRoot.[myReal*myReal + myImaginary*myImaginary]¶
  angle[ ]→
    Let theta ← Arcsine.[myImaginary/..magnitude[ ]].
    // ..magnitude[ ] is the result of sending magnitude[ ] to this Actor
    myReal>0 ◆
      True ˆ theta⊙
      False ˆ myImaginary >0 ◆
        True ˆ 180°-theta⊙43
        False ˆ 180°+theta ? ?¶
  plus[argument]→
    Let argumentRealPart ← argument.realPart[ ],
    argumentImaginaryPart ← argument.imaginaryPart[ ].
    Cartesian.[myReal+argumentRealPart,
    myImaginary+argumentImaginaryPart]¶
  times[argument]→
    Let argumentRealPart ← argument.realPart[ ],
    argumentImaginaryPart ← argument.imaginaryPart[ ].
    Cartesian.[myReal*argumentRealPart
    - myImaginary*argumentImaginaryPart,
    myImaginary*argumentRealPart
    + myReal*argumentImaginaryPart]¶
  equivalent[x] → // test if x is an equivalent complex number
    myReal=z.realPart[ ] ^ myImaginary=z.imaginaryPart[ ]§¶
```

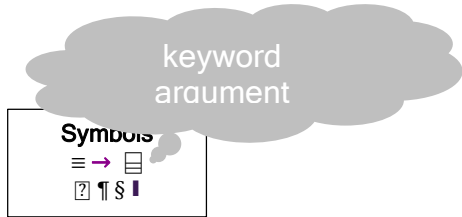
ⁱ Interface **Complex** with **realPart[]→ Float**,
imaginaryPart[]→ Float,
magnitude[]→ Float,
angle[]→ Degrees,
plus[Complex]→ Complex,
times[Complex]→ Complex,
equivalent[Complex]→ Boolean. ¶

Consequently,

- `Cartesian.[1, 2].realPart[]` is equivalent to `1`
- `Cartesian.[3, 4].magnitude[]` is equivalent to `5.0`
- `Cartesian.[0, 1].times[Cartesian.[0, 1]]` is equivalent to `Cartesian.[-1, 0]`⁴⁴

Arguments with named fields, i.e., `≡` and `:`

Polar Actors that implement **Complex** with named arguments **angle** and **magnitude** can be defined as follows:



```

Actor Polar[angle ≡ Degrees default 0°,
// angle of type Degrees is a named argument of Polar with
// default 0°
magnitude ≡ Length]
implements Complex using
angle[ ]→ angle⌘

realPart[ ]→ magnitude*Sine.[angle]⌘
imaginaryPart[ ]→ magnitude*Cosine.[angle]⌘
plus[argument]→
Cartesian.[argument.realPart[ ] + ..realPart[ ],
// ..realPart[ ] is the result of sending realPart[ ] to this Actor
argument.imaginaryPart[ ] + ..imaginaryPart[ ]]⌘
times[argument]→
Polar.[angle ≡ angle+argument.angle[ ],
magnitude ≡ magnitude*argument.magnitude[ ]]⌘
equivalent[x]→
..realPart[ ]=z.realPart[ ] ^ ..imaginaryPart[ ]=z.imaginaryPart[ ]$

```

Consequently,

- `Polar.[theAngle ≡ 0°, theMagnitude ≡ 1].realPart[]` is equivalent to `1`
- `(Polar.[theMagnitude ≡ 1]).equivalent[Cartesian.[1, 0]]` is equivalent to `True`

Lists, i.e., [] using Spread, i.e., [v]

A list expression begins with “**List**” followed by the type of list elementⁱ and expressions for list elementsⁱⁱ. Similarly “**Lists**” is used for a list of lists. The prefix operator “**v**” can be used to spread the elements of a list. For example

ⁱ delimited by < and >

ⁱⁱ delimited by “[” and “]”

- `List<Integer>[1, vList<Integer>[2, 3], 4]` is equivalent to `List<Integer>[1, 2, 3, 4]`.
- `Lists<Integer>[[1, 2], vList<Integer>[3, 4]]` is equivalent to `Lists<Integer>[[1, 2], 3, 4]`
- If `y` is `List<Integer>[5, 6]`, then `Lists<Integer>[1, 2, y, vy]` is equivalent to `Lists<Integer>[1, 2, [5, 6], 5, 6]`
- `List<Integer>[1, 2]` is the list of integers of type `Integer` with just 1 and 2.
- `List<Integer>[1, 2.0]` throws an exception because 2.0 is not of type `Integer`

The formal syntax of list expressions is in the following end note: 45.

A list pattern begins with “`List`” followed by the type of list elementⁱ and patterns for list elementsⁱⁱ. Within a list, “`v`” is used to match the pattern that follows with the list zero or more elements. Similarly “`Lists`” is used for a list of lists. For example:

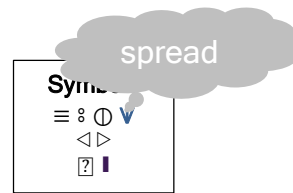
- `[[x, 2], vy]` is a pattern that matches `Lists<Integer>[[1, 2], 3, 4]` and binds `x` to 1 and `y` to `Lists<Integer>[3, 4]`
- `[[1, 2], v$$y]` is a pattern that only matches `Lists<Integer>[[1, 2], 3, 4]` if `y` is `Lists<Integer>[3, 4]`
- `[vx, vy]` is an illegal pattern because it can match ambiguously

The formal syntax of patterns is in the following end note: 46.

ⁱ delimited by `<` and `>`

ⁱⁱ delimited by “[” and “]”

As an example of the use of spread, the following procedure returns every other element of a list beginning with the first:



```

AlternateElements<aType>.[aList:List<aType>]:List<aType> ≡
aList ◆
[] : [] ⊕
[anElement] : [anElement] ⊕
[firstElement, secondElement] : [firstElement] ⊕
else :
[firstElement, secondElement, vremainingElements] :
[firstElement, vAlternateElements.[remainingElements]] ? |

```

Consequently,

- AlternateElements<Integer>.[[]] | is equivalent to List<Integer>[] |
- AlternateElements<Integer>.[[3]] | is equivalent to List<Integer>[3] |
- AlternateElements<Integer>.[[3, 4]] | is equivalent to List<Integer>[3] |
- AlternateElements<Integer>.[[3, 4, 5]] | is equivalent to List<Integer>[3, 5] |

Sets, i.e., { } using spreading, i.e., { v }

A set is an unordered structure with duplicates removed.

The formal syntax of sets is in the following end note: **47.**

Multisets, i.e., { } using spreading, i.e., { v }

A set is an unordered structure with duplicates allowed.

The formal syntax of multisets is in the following end note: **48.**

Maps, *i.e.*, `Map{ }`

A map is composed of pairs. For example `Map{ [3, "a"], ["x", "b"] }`!

Pairs in maps are unordered, *e.g.*, `Map{ [3, "a"], ["x", "b"] }` is equivalent to `Map{ ["x", "b"], [3, "a"] }`!

However, the expression `Map{ ["y", "b"], ["y", "a"] }` throws an exception because a map is univalent. As another example, for the contact records of 1.1 billion people, the following can compute a list of pairs from age to average number of social contacts of US citizens sorted by increasing age:

`Age ≡ Integer thatIs ≥0 ≤130`!

```
AgeToAverageOfNumberOfContactsPairsSortedByAge
  .[[records:Set<ContactRecord>]:List<[Age, Float]> ≡49
    records.filterii[[aRecord:ContactRecord] determinate →
      aRecord[[citizenship]] ⚡
        "US" : True⓪
        else : False?]
    .collectiii[[aRecord:ContactRecord] determinate →
      [aRecord[[yearsOld]],
       aRecord[[numberOfContacts]]]
    .reduceRangeiv
      [[aSetOfNumberOfContacts:Set<Integer>] determinate →
       aSetOfNumberOfContacts.averagev[ ]]
    .sortvi[LessThanOrEqual]!
```

ⁱ Structure `ContactRecord` `yearsOld` `≡ Age`,

`numberOfContacts` `≡ Integer`,

`citizenship` `≡ String`!

ⁱⁱ `Set<ContactRecord>` has `filter[[ContactRecord]→Boolean]→`
`Set<ContactRecord>`!.

ⁱⁱⁱ `Set<ContactRecord>` has
`collect [SingleArgumentToPair<ContactRecord, Age, Integer>]→`
`Map<Age, Set<Integer>>`!.

Interface `SingleArgumentToPair<Type1, Type2, Type3>` with
`[Type1]→ [Type2, Type3]`!.

^{iv} `Map<Age, Set<Integer>>` has
`reduceRange[FromTo<Set<Integer>, Float>]→ Map<Age, Float>`!.

Interface `FromTo<Type1, Type2>` with `[Type1]→ Type2`!.

^v `Set<Number>` has `average[]→ Float`!.

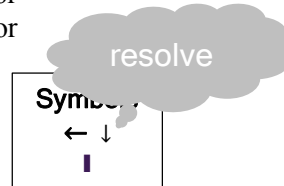
^{vi} `Map<Age, Float>` has `sort[PairTo<Age, Age, Boolean>]→ List<[Age, Float]>`!.

Interface `PairTo<Type1, Type2, Type3>` with `[Type1, Type2]→ Type3`!.

The formal syntax of maps is in the following end note: 50.

Futures, *i.e.*, Future and ↓

A future [Baker and Hewitt 1977] for an expression can be created in ActorScript by using “**Future**” preceding the expression. The operator “↓” can be used to “resolve” a future by returning an Actor computed by the future or throwing an exception. For example, the following expression is equivalent to `Factorial.[9999]`



```
Let aFuturei ← Future Factorial.[9999]。
↓aFuture // do not proceed until Factorial.[9999] has
           // resolvedii
```

Futures allow execution of expressions to be adaptively executed indefinitely into the future.⁵¹ For example, the following returns a future

```
Let aFuture ← Future Factorial.[9999],
    g ← ([afuture:Future<Integer>]:Integer → 5)。
           // g returns 5 regardless of its argument
g.[aFuture]
           // return 5 regardless of whether Factorial.[9999] has completediii
```

Note that the following are all equivalent⁵²:

- `↓Future (4+Factorial.[9999])`
- `4+↓Future Factorial.[9999]`
- `4+ⓈFactorial.[9999]`
- `Ⓢ(4+Factorial.[9999])`

Also `ⓈFactorial.[9999]+ⓈFibonacci.[9000]` is equivalent to the following:

```
Let niv ← ⓈFactorial.[9999],
    m ← ⓈFibonacci.[9000]。
n+m // return Factorial.[9999]+Fibonacci.[9000]
```

ⁱ f is of type `Future<Integer>`

ⁱⁱ *i.e.* returned or threw an exception

ⁱⁱⁱ *i.e.* `Factorial.[1000]` might not have returned or thrown an exception when 5 is returned. The future f will be garbage collected.

^{iv} n is of type `Integer`

In the following example, `Factorial.[9999]` might never be executed if `readCharacter.[]` returns the character 'x':

```

Let aFuture ← Future Factorial.[9999].
  readCharacter.[ ] ◆
    'x' ∶ 1⊕ // readCharacter.[ ] returned 'x'
  else ∶ 1+ ↓aFuture [?] |
    // readCharacter.[ ] returned something other than 'x'

```

In the above, program resolution of `aFuture` is highlighted in yellow.

The procedure `Size` below can compute the size of a `FutureList<String>`ⁱ concurrently with its being created:

```

Size.[aFutureList:FutureList<String>]:Integer ≡
  aFutureList ◆
  [] ∶ 0⊕
  [first, ↓rest] ∶ first.length[] + Size.[↓rest] [?] |
    // resolving a FutureList resolves only the head

```

Below is the definition of a procedure that computes a `FutureList` that is the “fringe” of the leaves of tree.ⁱⁱ

```

Fringe<aType>.[aTree:Tree<aType>]:FutureList<aType> ≡
  aTree ◆
  ∀Leaf<aType>[x] ∶ [x]⊕
  ∀Fork<aType>[tree1, tree2] ∶
    [↓Fringe.[tree1], ↓Postpone53 Fringe<aType>.[tree2]] [?] |

```

The above procedure can be used to define `SameFringe` that determines if two lists have the same fringe [Hewitt 1972]:

```

SameFringe<aType>
  .[aTree:Tree<aType>, anotherTree:Tree<aType>]:Boolean ≡
    // test if two trees have the same fringe
Fringe<aType>.[aTree] = Fringe<aType>.[anotherTree] |
  // = resolves futures in the fringes

```

ⁱ An instance of `FutureList<aType>` is *either*

1. the empty list of type `FutureList<aType>` *or*
2. a list whose first element is of `aType` and whose rest is of `Future<FutureList<aType>>`.

ⁱⁱ See definition of `Tree` above in this article.

The procedure below given a list of futures returns a **FutureList** with the same elements resolved:

```
FutureListOfResolvedElements<aType>
  .[aListOfFutures:List<Future<aType>>]:FutureList<aType> ≡
  aListOfFutures ◆
  [] § []⊕
  [aFirst, VaRest] §
  [↓aFirst,
   ↓Future FutureListOfResolvedElements<aType>. [↓aRest]] ?|
```

The formal syntax of futures is in the following end note: 54.

In-line Recursion (e.g., looping), i.e. [← , ←] ≡

Inline recursion (often called looping) is accomplished using an initial invocation with identifiers initialized using “←” followed by “≡” and the body.ⁱ

Below is an illustration of a loop Factorial with two loop identifiers n and accumulation. The loop starts with n equals 9 and value equal 1. The loop is iterated by a call to Factorial with the loop identifiers as arguments.

```
Factorial.[n ←9, accumulation ←1] ≡
  n ◆ 1 § accumulation⊕
  (> 1) § Factorial.[n-1, n* accumulation] ?|ii
```

The above compiles as a loop because the call to Factorial in the body is a “tail call” [Hewitt 1970, 1976; Steele 1977].

ⁱ This construct takes the place of **while**, **for**, *etc.* loops used in other programming languages.

ⁱⁱ equivalent to the following:

```
Factorial.[n:Integer ←9, accumulation:Integer ←1]:Integer ≡
  n ◆ 1 § accumulation⊕
  (> 1) § Factorial.[n-1, n* accumulation] ?|
```

The following expression returns a list of ten times successively calling the parameterless procedure P^i (of type $\text{To}\langle\text{Integer}\rangle^{\text{ii}}$):

```
FirstTenSequentially. $[n \leftarrow 10]:\text{List}\langle\text{Integer}\rangle \triangleq$ 
  n  $\diamond 1 \ni P.[ ] \oplus$ 
  (> 1)  $\ni \text{Let } x \leftarrow P.[ ] \bullet$ .
  [x,  $\forall$ FirstTenSequentially. $[n-1]$ ]  $\text{?}$   $\mathbf{!}$ 
```

The following returns one of the results of concurrently calling the procedure P^{iii} (of type $[] \rightarrow \text{Integer}$) ten times with no arguments:

```
OneOfTen. $[n \leftarrow 10]:\text{List}\langle\text{Integer}\rangle \triangleq$ 
  n  $\diamond 1 \ni P.[ ] \oplus$ 
  (> 1)  $\ni \text{P}.[ ] \text{ either } \text{OneOfTen}.\mathbf{[n-1]}$   $\text{?}$   $\mathbf{!}$ 
```

The formal syntax of looping is in the following end note: 55.

Strings

Strings are Actors that can be expressed using “**String**”, “[”, string arguments, and “]”. For example,

- **String**[‘1’, ‘23’, ‘4’] $\mathbf{!}$ is equivalent to “1234” $\mathbf{!}$.
- **String**[‘1’, ‘2’, ‘34’, ‘56’] $\mathbf{!}$ is equivalent to “123456” $\mathbf{!}$.
- **String**[**String**[‘1’, ‘2’], ‘34’] $\mathbf{!}$ is equivalent to “1234” $\mathbf{!}$.
- **String**[] $\mathbf{!}$ is equivalent to “” $\mathbf{!}$.

String patterns are delimited by “**String**”, “[” and “]”. Within a string pattern, “ \forall ” is used to match the pattern that follows with the list zero or more characters.

For example:

- **String**[x, ‘2’, \forall y] is a pattern that matches “1234” and binds x to ‘1’ and y to “34”.
- **String**[‘1’, ‘2’, \forall \$\$y] is a pattern that only matches “1234” if y is “34”.
- **String**[\forall x, \forall y] is an illegal pattern because it can match ambiguously.

ⁱ The procedure P may be indeterminate, *i.e.*, return different results on successive calls.

ⁱⁱ **Interface** $\text{To}\langle\text{aType}\rangle$ **with** $[] \rightarrow \text{aType}$. $\mathbf{!}$

ⁱⁱⁱ The procedure P may be indeterminate, *i.e.*, return different results on different calls.

As an example of the use of spread, the following procedure reverses a string:⁵⁶

```
Reverse.[aString:String]:String ≡
  aString ◊
  String[] ≡ String[] ⊕
  String[first, vrest] ≡ String[rest, first] ⊕
```

Symbols	
≡	⊕ ⊖
⊕	⊖
⊕	⊖

The formal syntax of string expressions is in the following end note: **57**.

General Messaging, *i.e.*, `.` and `⊕`

The syntax for general messaging is to use an expression for the recipient followed by “`.`” and an expression for the message.

For example, if `anExpression` is of type `Expression<Integer>` then,

```
anExpression.eval[anEnvironment]⊕
```

is equivalent to the following:

```
Let aMessagei ← eval⊕Expression<Integer>[anEnvironment].
  anExpression.aMessage⊕
```

The formal syntax of general messaging is in the following end note: **58**.

ⁱ aMessage:Message<Expression<Integer>>>

Language extension, *i.e.*, ()

The following is an illustration of language extension that illustrates postponed execution:⁵⁹

```
(("Postpone" anExpression:Expression <aType>):Postpone<aType> ≡  
  Actor implements Expression<Future<aType>> using  
    eval[anEnvironment]→  
      Future Actor implements aType using  
        aMessagei→ // aMessage received  
        Let postponed ← anExpression.eval[anEnvironment].  
        postponed.aMessage  
        // return result of sending aMessage to postponed  
        become postponed$!  
        // become the Actor postponed for  
        // the next message receivedii
```

The formal syntax of language extension is in the following end note: 60.

ⁱ aMessage:Message<aType>

ⁱⁱ this is allowed because postponed is of type aType

Atomic Operations, *i.e.* Atomic compare update updated notUpdated

For example, the following example implements a lockable that spins to lock:⁶¹

Actor **SpinLock[]** nonexclusive

locked := False. // initially unlocked

implements **Lockable**ⁱ using

lock[] → Attempt.[] ≜ // perform the loop Attempt as follows

Atomic locked compare False update True ⬢

// attempt to atomically update **locked** from **False** to **True**

updated ∃ **Preconditions locked = True.**

// **locked** must have contents **True**

Void ⊙ // if updated return **Void**

notUpdated ∃ Attempt.[] ? † // if not updated, try again

unLock[] →

Preconditions locked = True. // **locked** must have contents **True**

Void afterward locked := False § † // reset **locked** to **False**

Symbols

≡ → ⬢ ⊙
? † § †

The formal syntax of atomic operations is in the following end note: 62.

ⁱ Interface **Lockable** with **lock[]** → **Void**,
unLock[] → **Void**. †

Enumerations, i.e., Enumeration of using **Qualifiers, i.e.,** □

An enumeration provides symbolic names for alternatives using “**Enumeration** ” followed by the name of the enumeration, “**of**”, a list of distinct identifiers terminated by “.”.

For example,

**Enumeration DayName of Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday. ▮**

From the above definition, an enumerated day is available using a qualifier, e.g., **Monday**□**DayName**. Qualifiers provide structure for namespaces.

The formal syntax of qualifiers is in the following end note: **63**.

The procedure below computes the name of following day of the week given the name of any day of the week:

UsingNamespace DayName▮

FollowingDay.**[aDay:DayName]:DayName** ≡

aDay ◆ **Monday** : **Tuesday**,
 Tuesday : **Wednesday**,
 Wednesday : **Thursday**,
 Thursday : **Friday**,
 Friday : **Saturday**,
 Saturday : **Sunday**,
 Sunday : **Monday** □▮

The formal syntax of enumerations is in the following end note: **64**.

Native types, e.g., JavaScript, JSON, Java, and XML

Object can be used to create JavaScript Objects. Also, **Function** can be used to bind the reserved identifier **This**. For example, consider the following ActorScript for creating a JavaScript object aRectangle (with length 3 and width 4) and then computing its area 12:

```
Let aRectanglei ← Object {"length": 3, "width": 4},
    aFunction ← Function [] → This["length"] * This["width"],
    Rectangle["area"] := aFunction ●
    aRectangle["area"].[] |
```

The setTimeout JavaScript object can be invoked with a callback as follows that logs the string "later" after a time out of 1000:

```
setTimeout[] JavaScript.[1000,
    Function [] →
        console[] JavaScript.[ "log" ].["later"] |
```

JSON is a restricted version of **Object** that allows only Booleans, numbers, strings in objects and arrays.ⁱⁱ

Native types can also be used from Java. For example (s:String[]Java).substring[3, 5]ⁱⁱⁱ is the substring of s from the 3rd to the 5th characters inclusive.

Java types can be imported using **Import**, e.g.:

```
Namespace mynamespace |
Import java.math.BigInteger |
Import java.lang.Number |
```

After the above, **BigInteger.new["123"].instanceof[Number]** | is equivalent to **True** |.

ⁱ aRectangle is of type **Object**[]JavaScript

ⁱⁱ i.e. the following JavaScript types are not included in JSON: Date, Error, Regular Expression, and Function.

ⁱⁱⁱ **substring** is a method of the **String** class in Java

The following notation is used for XML.⁶⁵

```
XML <"PersonName"> <"First">"Ole-Johan" </"First">
<"Last"> "Dahl" </"Last"> </"PersonName">
```

and could print as:

```
<PersonName> <First> Ole-Johan </First>
<Last> Dahl </Last> </PersonName>
```

XML Attributes are allowed so that the expression

```
XML <"Country" "capital"="Paris"> "France" </"Country">
```

and could print as:

```
<Country capital="Paris"> France </Country>
```

XML construction can be performed in the following ways using the append operator:

- **XML** <"doc"> 1, 2, **V**[3] </"doc">] is equivalent to **XML** <"doc">1, 2, 3 </"doc">]
- **XML** <"doc">1, 2, **V**[3], **V**[4] </"doc">] is equivalent to **XML** <"doc"> 1, 2, 3, 4 </"doc">]

One-way messaging, e.g., \ominus , \leftarrow , and \rightarrow

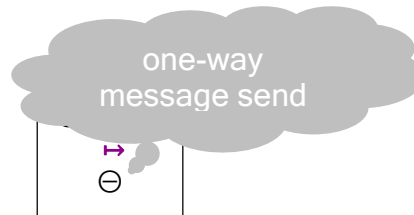
One-way messaging is often used in hardware implementations.

Each one-way named-message send consists of an expression followed by " \leftarrow ", a message name, and arguments delimited by "[" and "]".

The following is an interface for a customer that is used in request/response message passing for return type **aType**.⁶⁶

```
Interface Customer<aType> with  
  return[aType]  $\rightarrow$   $\ominus$ ,  
  throw[anException]  $\rightarrow$   $\ominus$ . I
```

For example, if aCustomer is of type **Customer<Integer>**, then 3 can be returned to aCustomer using aCustomer \leftarrow return[3].



The formal syntactic definition of one-way named-message sending is in the end note: **67**

Each one-way message handler implementation consists of a named-message declaration pattern followed by " \rightarrow " and a body for the response which must ultimately be " \ominus " which denotes no response.

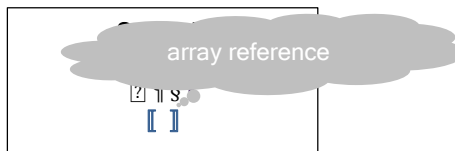
The formal syntactic definition of one-way named-message implementation is in the following end note: **68**

In the in-place implementation of QuickSort⁶⁹ (below), left is the index of the leftmost element of the subarray, right is the index of the rightmost element of the subarray (inclusive), and the number of elements in the subarray is right-(left+1).

```

QuickSort..[anArray:Array<Number>, left:Integer, right:Integer]:Void ≡
  Preconditions anArray.lower[ ] ≤ left ≤ right ≤ anArray.upper[ ].
  (left < right) ⇨ True : // If the array has 2 or more items
    Let pivotIndex ←
      Partition..[anArray, left, right, left+(right-left)/2] ●.
    Preconditions left ≤ pivotIndex ≤ right.
    ⓐ QuickSort..[anArray, left, pivotIndex-1],
      // Recursively sort elements smaller than the pivot
    ⓑ QuickSort..[anArray, pivotIndex+1, right] ⊕
      // Concurrently recursively sort elements at
      // least as big as pivot
  False : Void ? !

```



```

Partition..[anArray:Array<Number>, left:Integer, right:Integer,
  pivotIndex:Integer]:Integer ≡
  Preconditions anArray.lower[ ] ≤ left ≤ pivotIndex ≤ right ≤ anArray.upper[ ].
  Let pivot ← anArray[pivotIndex] ●.
  anArray.swap[pivotIndex, right] ●70
  Let finalStoreIndex ←
    Move..[iterationIndex:Integer ← left,
      storeIndex:Integer ← left]:Integer ≜
    Preconditions left ≤ storeIndex ≤ iterationIndex ≤ right.
    iterationIndex < right ⇨
      True :
        anArray[iterationIndex] ≤ pivotValue ⇨
          True :
            anArray.swap[iterationIndex, storeIndex] ●
            Move..[iterationIndex+1, storeIndex+1] ⊕
          False : Move..[iterationIndex+1, storeIndex] ? ⊕
      False : storeIndex ? ●.
  anArray.swap[finalStoreIndex, right] ● // Move Actor to its final place
  finalStoreIndex. !

```

For example, consider the following example:

```

Let anArray ← Array.[3, 2, 1].
QuickSort..[anArray, 0, 1] ●
anArray. !

```

The above returns `Array[1, 2, 3]`!

Appendix 2: Meta-circular definition of ActorScript

It might seem that a meta-circular definition is a strange way to define a programming language. However, as shown in the references, concurrent programming languages are not reducible to logic. Consequently, an augmented meta-circular definition may be one of the best alternatives available.

The message `eval`

John McCarthy is justly famous for Lisp. One of the more remarkable aspects of Lisp was the definition of its interpreter (called Eval) in Lisp itself. The exact meaning of Eval defined in terms of itself has been somewhat mysterious since, on the face of it, the definition is circular.⁷¹

The basic idea is to send an expression an `eval` message with an environment to instead of the Lisp approach of send the procedure Eval the expression and environment as arguments.

Each `eval` message has an environment with the bindings of program identifiers:

```
Interface Expression<aType> with
    eval[Environment] → aType. !
```

The tokens (and) are used to delimit program syntax.

```
((anIdentifier: Identifier<aType>): Expression<aType> ≡
    eval[anEnvironment] → anEnvironment.lookup[anIdentifier] !
```

The interface `Type` implements `isExtension`

The interface `Type` has message `isExtension`:

```
Interface Type with isExtension[aType: Type] → Boolean. !
```

```
((anotherType: Type<anotherType> "∃" aType: Type<aType>)
    : Expression<Boolean> ≡
    eval[anEnvironment] →
    (anotherType.eval[anEnvironment])
    .isExtension[aType.eval[anEnvironment]] !
```

Future, ↓, and ⊕

The interface **Future** is used for futures:

Interface Future<aType> with **resolve**[] → aType. **!**

```
(("Future" anExpression:Expression<aType>))
                                     :Expression<Future<aType>> ≡
Actor implements Expression<Future<aType>> using
eval[anEnvironment] →
  Let aFuture:Future<aType> ←
    Future Try anExpression.eval[anEnvironment]
      catch ⚡
        anException :
          Actor
            implements Future<aType>
              resolve[ ] → Throw anException. ?
  Actor implements Future<aType> using
    resolve[ ] → ↓aFuture §!
```

```
(("↓" anExpression:Expression<Future<aType>>))
                                     :Expression<aType> ≡
Actor implements Expression<aType> using
eval[anEnvironment] →
  anExpression.eval[anEnvironment].resolve[ ] §!
```

```
(("⊕" anExpression:Expression<aType>))
                                     :Expression<aType> ≡
Actor implements Expression<aType> using
eval[anEnvironment] →
  ↓Future anExpression.eval[anEnvironment] §!
```

The message **match**

Patterns are analogous to expressions, except that they take receive match messages:

Interface Pattern<aType> with
match[aType, Environment] → **Nullable**<Environment>,
mustMatch[aType, Environment] → **Environment**. **!**

```
(anIdentifier: Identifier <aType>): Pattern <aType> ≡  
  match[anActor: aType, anEnvironment] →  
    anEnvironment.bind[anIdentifier, to ⊞ anActor] ■
```

```
("_"): UniversalPattern <aType> ≡  
  match[anActor: aType, anEnvironment] → anEnvironment ■
```

```
("$$" anExpression: Expression <expressionType>)  
  : ValuePattern <aType> ≡  
  match[anActor: aType, anEnvironment] →  
    (anActor = anExpression.eval[anEnvironment]) ◆  
    True : Nullable <Environment> [anEnvironment] ⊕  
    False : Nullable <Environment> [Null] ⊗ ■
```

Message sending, e.g., ▪

```
(procedure: Expression <argumentsType> → returnType>
  "▪" "[" arguments: Arguments <argumentsType> "]"
  ":" returnType)
  : ProcedureSend <argumentsType, returnType> ≡
eval[anEnvironment] →
  (procedure. eval[anEnvironment])
  ▪ [V(expressions. eval[anEnvironment])] §!
```

```
(recipient: Expression <recipient> "▪"
  name: MessageName "[" Varguments
    : Arguments <argumentsType> "]" )
  : NamedMessageSend <expressionType> ≡
eval[anEnvironment] →
  Let aRecipient ← recipient. eval[anEnvironment].
  aRecipient
  ▪ Message[QualifiedName[name recipientType],
    [Varguments. eval[anEnvironment]]] §!
```

```
(recipient: Expression <recipientType>
  "▪" aMessage: Message <Message <recipientType>>>)
  : UnnamedMessageSend <expressionType> ≡
eval[anEnvironment] →
  (recipient. eval[anEnvironment])
  ▪ (aMessage. eval[anEnvironment]) §!
```

List Expressions and Patterns

```

([" firstExpression:Expression<type1>
  "," secondExpression:Expression<type2> "]" )
  :ListExpression <expressionType> ≡
eval[anEnvironment]→
  Let first ← firstExpression.eval[anEnvironment],
      second ← secondExpression.eval[anEnvironment]。
  [first, second]§!

```

```

([" firstExpression:Expression<type1>
  "," "V" restExpression:Expression<type2> "]" )
  :ListExpression <expressionType> ≡
eval[anEnvironment]→
  Let first ← firstExpression.eval[anEnvironment],
      rest ← restExpression.eval[anEnvironment]。
  [first, Vrest]§!

```

```

([" firstPattern:Pattern<aType>
  "," "V" restPattern:ListPattern<aType> "]" )
  :ListPattern <aType> ≡
match[anActor:aType, anEnvironment]→
  anActor ◆
  [first, Vrest] :
    firstPattern.match[first, anEnvironment] ◆
    ∇Null Environment :
      Nullable<Environment>[Null],
      aNewEnvironmentΔEnvironment :
        restPattern.match[restValue, aNewEnvironment] ?①
  else : Nullable<Environment>[Null] ?§!

```


Exceptions

```

(("Try" anExpression: Expression <aType>
  "catch" exceptions: ExpressionCases <Exception, aType> "?" )
  : TryExpression <aType> ≡

eval[anEnvironment] →
  Try anExpression.eval[anEnvironment] catch
  anException: Exception ∋ CasesEval.[anException,
    exceptions,
    anEnvironment] ?$!

```

```

(("Try" anExpression: Expression <aType>
  "cleanup" aCleanup: Expression <aType> )
  : TryExpression <aType> ≡

eval[anEnvironment] →
  Try anExpression.eval[anEnvironment] catch
  _ ∋ aCleanup.eval[anEnvironment] ●
  Rethrow ?$!

```

Continuations using perform

A continuation is a generalization of expression for executing in cheese, which receives **perform** messages:

Interface Continuation <aType> with
perform [Environment, CheeseQ] → aType. |

Discrimination Construct <aType> between
Continuation <aType>,
Expression <aType>. |

```

Execute.[aConstruct: Construct <aType>,
  anEnvironment: Environment,
  aCheeseQ: CheeseQ] ≡
  aConstruct ◆ aContinuation: Continuation <aType> ∋
    aContinuation.perform[anEnvironment,
      aCheeseQ] ⊕
  anExpression: Expression <aType> ∋
    anExpression.eval[anEnvironment] ?$!

```

Atomic compare and update

```

(("Atomic" location: Expression<Location<anotherType>,
  "compare" comparison: Expression<anotherType>
  "update" update: Expression<anotherType> "⚡"
  "updated" "⚡"
    compareIdentical: ContinuationList<aType> "⊕"
  "notUpdated" "⚡"
    compareNotIdentical: ContinuationList<aType>))
  :Atomic<aType> ≡

perform[anEnvironment, aCheeseQ]→
(location. eval[anEnvironment])
  .compareAndConditionallyUpdate[comparison. eval[anEnvironment],
    update. eval[anEnvironment]] ⚡
  True ⚡ compareIdentical. perform[anEnvironment, aCheeseQ] ⊕
  False ⚡
    compareNotIdentical. perform[anEnvironment, aCheeseQ] ?|

Actor SimpleLocation<anotherType> [initialContents]
  contents := initialContents,
  implements Location<anotherType> using
    compareAndConditionallyUpdate[comparison, update]→
      (contents = comparison) ⚡
      True ⚡ True afterward contents := update ⊕
      False ⚡ False ?|

```

Cases

```

(anExpression: Expression <anotherType> “⊖”
  cases: ExpressionCases <anotherType, aType> “?”)
  : CasesExpression <aType> ≡

eval[anEnvironment] →
  CasesEval.[anExpression.eval[anEnvironment],
    cases,
    anEnvironment] $!

CasesEval.[anActor: anotherType,
  cases: List <ExpressionCase <anotherType, aType>>,
  anEnvironment: Environment] ≡

cases ⊖
  [] : Throw NoApplicableCase[ ],
  [first, Vrest] :
    first ⊖ (aPattern: Pattern <anotherType> “:”
      anExpression: Expression <aType>)
      : ExpressionCase <aType> :
    aPattern.match[anActor, anEnvironment] ⊖
    VNull Environment :
      CasesEval.[anActor, rest, anEnvironment] ⊖
      newEnvironment Δ Environment :
      anExpression.eval[newEnvironment] ? ⊖
      (“else” elsePattern: Pattern <anotherType> “:”
        elseExpression: Expression <aType>)
      : ExpressionElseCase <aType> :
    elsePattern.match[anActor, anEnvironment] ⊖
    VNull Environment :
      Throw ElsePatternMustMatch[ ] ⊖
      newEnvironment Δ Environment :
      elseExpression.eval[newEnvironment] ? ⊖
      (“else” “:”
        elseExpression: Expression <aType>)
      : ExpressionElseCase <aType> :
    elseExpression.eval[anEnvironment] ⊖
  else : Throw NoApplicableCase[ ] ? ? !

```

```

((anExpression: Expression <anotherType> “◆”
  cases: ContinuationCases <anotherType, aType> “?”)
  : CasesContinuation <aType> ≡
perform[anEnvironment, aCheeseQ] →
  CasesPerform.▪[anExpression.▪eval[anEnvironment], cases,
    anEnvironment, aCheeseQ] $ !
CasesPerform.▪[anActor: anotherType,
  cases: List<ContinuationCase <aType>>,
  anEnvironment: Environment,
  aCheeseQ: CheeseQ] ≡
cases ◆
  [] ⊗ Throw NoApplicableCase[],
  [first, ▼rest] ⊗
    first ◆ ((aPattern: Pattern <anotherType> “⊗”
      aContinuation: Continuation <aType>))
      : ContinuationCase <aType> ⊗
      aPattern.▪match[anActor, anEnvironment] ◆
      ∀Null Environment ⊗
      CasesPerform.▪[anActor,
        rest,
        anEnvironment,
        aCheeseQ] ⊕
      newEnvironment Δ Environment ⊗
      aContinuation.▪perform[newEnvironment,
        aCheeseQ] ? ⊕
    (“else”
      elsePattern: Pattern
        <anotherType> “⊗”
        elseContinuation: Continuation <aType>))
        : ContinuationElseCase <aType> ⊗
      elsePattern.▪match[anActor, anEnvironment] ◆
      ∀Null Environment ⊗
      Throw ElsePatternMustMatch[] ⊕
      newEnvironment Δ Environment ⊗
      elseContinuation.▪eval[newEnvironment] ? ⊕
    (“else” “⊗”
      elseContinuation: Continuation <aType>))
      : ContinuationElseCase <aType> ⊗
      elseContinuation.▪perform[anEnvironment, aCheeseQ] ⊕
    else ⊗ Throw NoApplicableCase[] ? ? !

```

Holes in the cheese

```
((anExpression: Expression <aType>
  "afterward" someAssignments: Assignments ".")
  : Continuation <aType> ≡
  perform[anEnvironment, aCheeseQ] →
  Let anActor ← anExpression.eval[anEnvironment] ●。
  someAssignments.carryOut[anEnvironment, aCheeseQ] ●
  aCheeseQ.leave[ ] ●
  anActor. §I
```

```
((aVariable: Variable <aType>
  "!=" anExpression: Expression <aType>): Assignment ≡
  carryOut[anEnvironment] →
  anEnvironment.assign[aVariable,
    to ≡ anExpression.eval[anEnvironment]] §I
```

```
("Hole" anExpression: Expression <aType>): Hole <aType> ≡
  perform[anEnvironment, aCheeseQ] →
  aCheeseQ.leave[ ] ●
  anExpression.eval[anEnvironment.freeze[ ] ] §I
```

```
("Hole" anExpression: Expression <aType>
  "after"
  aPreparation: Preparation): Continuation <aType> ≡
  perform[anEnvironment, aCheeseQ] →
  Let frozenEnvironment ← anEnvironment.freeze[ ] ●。
  // create frozen environment so that
  // preparation does not affect evaluating anExpression
  aPreparation.carryOut[anEnvironment, aCheeseQ] ●
  aCheeseQ.leave[ ] ●
  anExpression.eval[frozenEnvironment]. §I
```

```

(("Hole" anExpression: Expression <anotherType>
 "afterward" anAfterward: AfterwardContinuation <aType> "[?]")
      :Continuation<aType> ≡
perform[anEnvironment, aCheeseQ]→
  Let frozenEnvironment ← anEnvironment.freeze[ ] ●。
  aCheeseQ.leave[ ] ●
  Try Let anActor ← anExpression.eval[frozenEnvironment] ●。
    aCheeseQ.enter[ ] ●
    anAfterward.perform[anEnvironment, aCheeseQ] ●
    anActor afterward aCheeseQ.leave[ ]。
  catch ⚡
    anException: ApplicationException :
      aCheeseQ.enter[ ] ●
      anAfterward.perform[anEnvironment, aCheeseQ] ●
      throw anException afterward aCheeseQ.leave[ ] [?]。 §I

```

```

(("Hole" anExpression: Expression <anotherType>
 "returned" ⚡
  returnedCases: ContinuationCases <anotherType, aType> "[?]"
 "threw" ⚡
  threwCases: ContinuationCases <anotherType, aType> "[?]")
      :Continuation<anotherType, aType> ≡
perform[anEnvironment, aCheeseQ]→
  Let frozenEnvironment ← anEnvironment.freeze[ ] ●。
  aCheeseQ.leave[ ] ●
  Try Let anActor ← anExpression.eval[frozenEnvironment] ●。
    aCheeseQ.enter[ ] ●
    CasesPerform.perform[anActor,
      returnedCases,
      anEnvironment,
      aCheeseQ]。
  catch ⚡ anException: ApplicationException :
    aCheeseQ.enter[ ] ●
    CasesPerform.perform[anException,
      threwCases,
      anEnvironment,
      aCheeseQ] [?]。 §I

```

```

(("Enqueue" anExpression:QueueExpression "●")
  :Enqueue <aType> ≡
perform[anEnvironment, aCheeseQ]→
  Let anInternalQ ← anExpression.eval[anEnvironment]。
  anInternalQ.enqueueAndLeave[ ] §!

```

```

(("Enqueue" anExpression:QueueExpression "●"
  aContinuation:Continuation <aType>):Enqueue <aType> ≡
perform[anEnvironment, aCheeseQ]→
  Let anInternalQ ← anExpression.eval[anEnvironment]。
  anInternalQ.enqueueAndLeave[ ] ●
  aContinuation.perform[anEnvironment, aCheeseQ]。 §!

```

A Simple Implementation of Actor

The implementation below does not implement queues, holes, and relaying.

```

(("Actor" declarations:ActorDeclarations
  "implements" Identifier <aType>
  "using" handlers:Handlers <anInterface> "$"):Actor <aType> ≡
Actor implements Expression <anInterface> using
eval[anEnvironment]→
  Initialized.[anInterface.eval[anEnvironment],
    handlers,
    declarations.initialize[anEnvironment],
    SimpleCheeseQ.[ ]] §!

```

```

Initialized.[anInterface:aType,
             handlers:List<Handler<aType>>,
             anEnvironment:Environment,
             cheeseQ:CheeseQ]:aType ≡
Actor implements anInterface using
receivedMessage:Message<aType> →
// receivedMessage received for anInterface
aCheeseQ.enter[ ] ●
Let aReturned ← Try Select.[receivedMessage,
                             handlers,
                             anEnvironment,
                             aCheeseQ]
                             cleanup aCheeseQ.leave[ ] ●。
// leave cheese and rethrow exception
aCheeseQ.leave[ ] ●
aReturned. §!

```

```

Select.[receivedMessage:Message<aType>,
        handlers:List<Handler<aType>>,
        anEnvironment:Environment,
        aCheeseQ:CheeseQ]:aType ≡
handlers ◆
[ ] § Throw NotApplicable [ ] ⊕
[(aMessageDeclaration:MessageDeclaration<aType> "→"
  body:Continuation<aType>)]
  :ContinuationHandler<aType> ⊕
∨restHandlers] §
aMessageDeclaration.match[receivedMessage,
                           anEnvironment] ◆
∇Null Environment § Select.[receivedMessage,
                              restHandlers,
                              anEnvironment,
                              aCheeseQ] ⊕
// process next handler
newEnvironmentΔEnvironment §
Execute.[body, newEnvironment, aCheeseQ] [?] [?] !
// execute body with extension of anEnvironment

```


An implementation of cheese that never holds a lock

The following is an implementation of cheese that does not hold a lock.⁷²

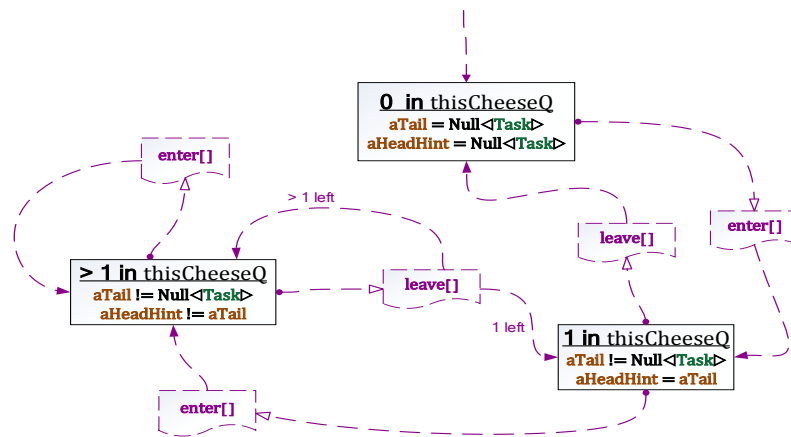
```

Actor SimpleCheeseQ[ ] nonexclusive
  invariants aTail = Nullable<Activity>[Null]
    ⇒ previousToTail = Nullable<Activity>[Null]。
  aHeadHint := Nullable<Activity>[Null], // aHeadHint: Nullable<Activity>[Null]73
  aTail := Nullable<Activity>[Null]。 // aTail: Nullable<Activity>[Null]74
implements CheeseQ75 using
  enter[ ] in myActivity →76
    Preconditions myActivity[previous] = Nullable<Activity>[Null],
      myActivity[nextHint] = Nullable<Activity>[Null]。
    attempt.[ ]:Void ≜
      myActivity[previous] := aTail ● // set provisional tail of queue
      Atomic aTail compare aTail update myActivity ◆
        updated : // inserted myActivity in cheese queue with previous
          myActivity[previous] = Nullable<Activity>[Null] ◆
          True : Void ⊙ // successfully entered cheese
          False : Suspend [ ] ⊙ // current activity is suspended
        notUpdated : attempt.[ ] [ ] ⊙ // make another attempt
    leave[ ] in myActivity → // leave message received running myActivity
    Preconditions aTail != Nullable<Activity>[Null]。77
    Let ahead ← [SubCheeseQ]head[ ] ●。
    Preconditions ahead = myActivity。
    Atomic aTail compare ahead update Nullable<Activity>[Null] ◆
      updated : // last activity has left this cheese queue
        aHeadHint := Nullable<Activity>[Null] ●
        Void ⊙
      notUpdated : // another activity is in this cheese queue
        aHeadHint := ahead[nextHint] ●
        MakeRunnable aHeadHintΔActivity[ ]$
  also implements SubCheeseQ78 using
  head[ ] → Preconditions aTail != Nullable<Activity>[Null]。
  findHead.[backIterator:Activity ←
    aHeadHint = Nullable<Activity>[Null] ◆
    True : aTailΔActivity ⊙
    False : aHeadHintΔActivity [ ]:Activity ≜
  backIterator[previous] ◆
  ∀Null Activity : // backIterator is head of this cheese queue
    aHeadHint := Nullable backIterator ●
    backIterator. ⊙
  previousBackIteratorΔActivity :
    // backIterator is not the head of this cheese queue
    previousBackIterator[nextHint] := Nullable backIterator ●
    // set nextHint of previous to backIterator
  findHead.[previousBackIterator]. [ ]$

```

The algorithm used in the implementation of **CheeseQ** above is due to Blaine Garst [private communication] cf. [Ladan-Mozes and Shavit 2004].

There is a state diagram for the implementation below:



```

Actor SimpleInternalQ [aCheeseQ:CheeseQ] nonexclusive
  aHead := Nullable<Activity>[Null],      // aHead:Nullable<Activity>
  aTail := Nullable<Activity>[Null]。
  implements InternalQ79 using
    enqueueAndLeave[] in myActivity →
      // enqueueAndLeave message received in myActivity
      [:SubInternalQ.remove[myActivity]]●
      aCheeseQ.leave[]● // myActivity is the head of aCheeseQ
      Suspend⌈
        // myActivity is suspended and when resumed returns Void ⌈
      enqueueAndDequeue[anInternalQ] in myActivity →
        Preconditions ¬anInternalQ.isEmpty[]。
        [:SubInternalQ.add[myActivity]]●
        .dequeue[]●
        Suspend。⌈
      dequeue[] in myActivity → Preconditions ¬.isEmpty[]。
      aCheeseQ.leave[]● // myActivity is the head of aCheeseQ
      MakeRunnable [:SubInternalQ.remove[]]⌈
        // make runnable the removed activity
      isEmpty[] → aTail = Nullable<Activity>[Null]§
  also implements SubInternalQ80 using
    add[anActivity] →
      aTail ⋄
      ∀Null Activity :
        Void afterward aHead := Nullable anActivity,
          aTail := Nullable anActivity。⊙
      theTailΔActivity : Void afterward theTail[[rest]] := anActivity [?]⌈
    remove[] → Preconditions ¬.isEmpty[]。
    Let theFirst ← aHeadΔActivity●。
    aTail=aHead ⋄
    True : theFirst afterward aHead := Nullable<Activity>[Null],
      aTail := Nullable<Activity>[Null]。⊙
    False : theFirst afterward aHead := (aHeadΔActivity)[rest] [?]§!

```

Appendix 3. Inconsistency Robust Logic Programs

Logic Programs⁸¹ can logically infer computational steps.

Forward Chaining

Forward chaining is performed using \vdash

((\vdash *Theory* *PropositionExpression*))
Assert *PropositionExpression* for *Theory*.

((**When** (\vdash *Theory* *aProposition:Pattern* \rightarrow *Expression*))
When *aProposition* holds for *Theory*, evaluate *Expression*.

Illustration of forward chaining:

\vdash_t Human[Socrates]||

When \vdash_t Human[x] \rightarrow \vdash_t Mortal[x]||

will result in asserting Mortal[Socrates] for theory t

Backward Chaining

Backward chaining is performed using \Vdash

((\Vdash *Theory* *aGoal:Pattern* \rightarrow *Expression*))
Set *aGoal* for *Theory* and when established evaluate *Expression*.

((\Vdash *Theory* *aGoal:Pattern*):*Expression*)
Set *aGoal* for *Theory* and return a list of assertions that satisfy the goal.

((**When** (\Vdash *Theory* *aGoal:Pattern* \rightarrow *Expression*))
When there is a goal that matches *aGoal* for *Theory*, evaluate *Expression*.

Illustration of backward chaining:

$\vdash_t \text{Human}[\text{Socrates}] \mathbf{!}$

When $\Vdash_t \text{Mortal}[x] \rightarrow (\Vdash_t \text{Human}[\$x] \rightarrow \vdash_t \text{Mortal}[x]) \mathbf{!}$

$\Vdash_t \text{Mortal}[\text{Socrates}] \mathbf{!}$

will result in asserting $\text{Mortal}[\text{Socrates}]$ for theory t .

SubArguments

This section explains how subargumentsⁱ can be implemented in natural deduction.

When $\Vdash_s (\psi \vdash_t \phi) \rightarrow$

Let $t' \leftarrow \text{extension} \cdot [t] \circ$

$\vdash_{t'} \psi,$

$\Vdash_{t'} \phi \rightarrow \Vdash_s (\psi \vdash_t \phi) \circ \mathbf{!}$

Note that the following hold for t' because it is an extension of t :

- **when** $\vdash_t \theta \rightarrow \vdash_{t'} \theta \mathbf{!}$
- **when** $\Vdash_{t'} \theta \rightarrow \Vdash_t \theta \mathbf{!}$

ⁱ See appendix on Inconsistency Robust Natural Deduction.

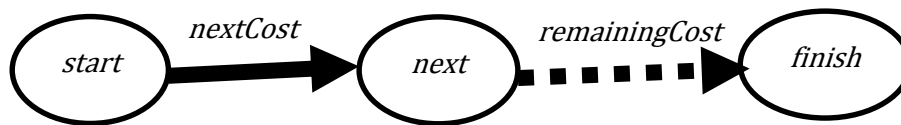
Aggregation using Ground-Complete Predicates

Logic Programs in ActorScript are a further development of Planner. For example, suppose there is a ground-complete predicate⁸² `Link[aNode, anotherNode, aCost]` that is true exactly when there is a path from `aNode` to `anotherNode` with `aCost`.

```
When ⊢ Path[aNode, aNode, aCost] →
    // when a goal is set for a cost between aNode and itself
    ⊢ aCost=0 ⊢ // assert that the cost from a node to itself is 0
```

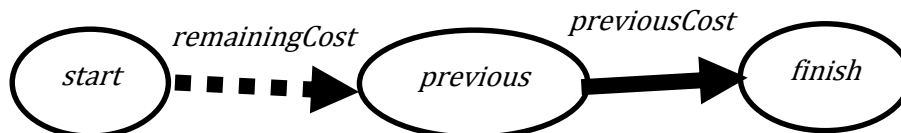
The following goal-driven Logic Program works forward from `start` to find the cost to `finish`:⁸³

```
When ⊢ Path[start, finish, aCost] →
    ⊢ aCost=Minimum {nextCost + remainingCost
        | ⊢ Link[start, next≠start, nextCost],
        Path[next, finish, remainingCost]} ⊢
    // a cost from start to finish is the minimum of the set of the sum of the
    // cost for the next node after start and
    // the cost from that node to finish
```



The following goal-driven Logic Program works backward from `finish` to find the cost from `start`:

```
When ⊢ Path[start, finish, aCost] →
    ⊢ aCost=Minimum {remainingCost + previousCost
        | ⊢ Link[previous≠finish, finish, previousCost],
        Path[start, previous, remainingCost]} ⊢
    // the cost from start to finish is the minimum of the set of the sum of the
    // cost for the previous node before finish and
    // the cost from start to that Node
```



Note that all of the above Logic Programs work together concurrently providing information to each other.

Appendix 4. ActorScript Symbols with Readings. IDE ASCII, and Unicode code points

Symbol	IDE ASCII ⁱ	Read as	Category	Matching Delimiters	Unicode (hex)
█	;;	end	top level terminator		32DA
:	:	of exact type	infix		
⊠	[:]	cast this actor to	prefix		2360
Δ	/ _ \	expression/ pattern discriminate	infix		2206
∇	\ - /	structure discriminate	prefix		2207
↓	\ /	resolve	prefix		2139
⊠	[.]	qualified by	infix		22A1
▪	.	is sent	infix		
▪▪	..	delegate to this Actor	prefix		
Ⓜ	()	concurrently ⁸⁴	prefix		29B7
↪	- >	message type returns ⁸⁵	infix		21A6
→	-- >	message received ⁸⁶		¶	2192
←	< --	be ⁸⁷	infix		2190
⊠	?	cases	separator	?	FFFD
Ⓜ	()	another case	separator	⊠ and ?	29B6
?	[?]	end cases	terminator	⊠ and catch⊠	2370
¶	\ p	another message handler	separator for handlers	→	00B6
§	\ s	end handlers	terminator	implements	00A7
⋈	(:)	case	separator for case		2982
●	\ _ /	before	separator	Let bindings, Do preparations, Enqueue,	00C4
◦	()	end	terminator	Do expressions and ⋈	FF61
≡	== ⁸⁸	defined as	infix		2261
≐	= / \ =	to be	infix		225C
≑	: =	is assigned	infix		2254
\$\$	\$\$	matches value of ⁸⁹	prefix		
=	=	same as?	infix		
⊠	[=]	keyword or field	infix		2338

ⁱ These are only examples. They can be redefined using keyboard macros according to personal preference.

$\text{:}\equiv$	$\text{:} [=]$	assignable field	infix		
\triangleleft	$\langle $	begin type parameters	left delimiter	\triangleright (Unicode hex: 0077)	0076
\forall	$\backslash /$	spread ⁹⁰	prefix		2A5B
$\{$	$\{$	begin set	left delimiter	$\}$	
$[$	$[$	begin list	left delimiter	$]$	
$\{$	$\{ $	begin multi-set	left delimiter	$\}$	2983
\llbracket	\llbracket	array reference	left delimiter	\rrbracket (Unicode hex: 27E7)	27E6
$($	$($	begin grouping	left delimiter	$)$	
$($	$($	begin syntax	left delimiter	$)$ (Unicode hex: 2986)	2985
\ominus	$(-)$	nothing ⁹¹	expression		229D
\leftarrow	\llleftarrow	one-way send	infix		219E
\rightarrow	\rrrightarrow	one-way receive	infix	\Uparrow	21A0
\sqcup	$\sqcup $	join	infix		2294
\sqsubseteq	$[\leq]$	constrained by	infix		2291
\sqsupseteq	$[\geq]$	extends	infix		2292
\Rightarrow	\Rightarrow	logical implication	infix		21E8
\Leftrightarrow	\Leftrightarrow	logical equivalence	infix		21D4
\wedge	\wedge	logical conjunction	infix		00D9
\vee	\vee	logical disjunction	infix		00DA
\neg	$- $	logical negation	prefix		00D8
\vdash	$ -$	assert	prefix and infix		22A2
\Vdash	$\ -$	goal	prefix and infix		22A9
$//$	$//$	begin 1-line comment	prefix	EndOfLine	
$/*$	$/*$	begin comment	prefix	$*/$	

Type Discrimination, *i.e.*, Discrimination, Δ and ∇

```

(("Discrimination" aDiscrimination "of"
  typeExpressions: Expressions <Type> "."): Definition ≡
  Actor implements Definition using
  eval[anEnvironment] →
  Let types: List <Type> ←
    typeExpressions. eval[anEnvironment].
  Let aDiscrimination[aType: Type] ≡
    aType ∈ types ◆
    True : DiscriminationInstance. [x, aType] ⊕
    False : Throw NotADiscriminant[aType] ?
  DiscriminationInstance. [x: aType, aType: Type] ≡
  Actor implements aDiscrimination using
  discriminate[anotherType] →
    anotherType ◆
    aType : x ⊕
    else :
      Throw WrongDiscriminant[anotherType] ?
  (discrimination: Expression <aDiscrimination>
    "Δ" discriminant: Expression <Type>)
    : Expression <discriminant> ≡
  Actor implements Expression <discriminant> using
  eval[anEnvironment] →
    (discrimination. eval[anEnvironment])
    .discriminate[discrimination
      .eval[anEnvironment]],
  (aPattern: Pattern <aType>
    "Δ" discriminant: Expression <Type>))
    : Pattern <aDiscrimination> ≡
  Actor implements Pattern <aDiscrimination> using
  match[anActor: aType, anEnvironment] →
    aPattern. match[anActor
      Δ(discriminant
        .eval[anEnvironment]),
      anEnvironment].

  Void!

```

End Notes

¹ Quotation by the author from late 1960s.

² to use a reserved word as an identifier it could be prefixed, e.g., `_actor`

³ The delimiters `(` and `)` are used to delimit program syntax with the character `"` and the character `"` to delimit tokens. For example, `(3 "+" 4)` is an expression that can be evaluated to 7. A special font is used for syntactic categories.

For example,

```
((x:Numerical "+" y:Numerical):Numerical)
Numerical ≡ Expression
```

Also,

```
((Numerical "-" Numerical):Numerical)
("-" Numerical):Numerical
(Numerical "*" Numerical):Numerical
(Numerical "/" Numerical):Numerical
("Remainder" Numerical "/" Numerical):remainder:Numerical
("QuotientRemainder" Numerical "/" Numerical)
:[Numerical, Numerical]
("True" ⊔ "False"):Expression <Boolean>
(Expression <Boolean> "∧" Expression <Boolean>)
:Expression <Boolean>
(Expression "∨" Expression):Expression <Boolean>
("¬" Expression <Boolean>):Expression <Boolean>
("Throw" Expression):Expression
```

⁴ See explanation of syntactic categories above. A word must begin with an alphabetic character and may be followed by one or more numbers and alphabetic characters.

```
Identifier ≡ Word ≡ Expression
```

// an *Identifier* is a *Word*, which is a subcategory of *Expression*

```
((Expression ⊔ Definition ⊔ Judgment)) "I"):Top
```

⁵ `Type ≡ Expression <Type>`

```
(( aType:Type "→" anotherType:Type ):Type)
```

```
("[" Types "]" ):Type
```

```
( ⊔ MoreTypes ):Types
```

```
(Type ⊔ (Type "," MoreTypes)):MoreTypes
```

⁶ `(Identifier <aType> (⊔ (":" Type <aType>))`

```
"≡" ExpressionList <aType>):Definition <Identifier>
```

```

(((Expression <aType> ( ⊔ “.” )))
  ⊔ (Expression (“,” ⊔ “●”) MoreExpressionList <aType>))
                                     :ExpressionList <aType> |

(((Expression <aType> “.”)
  ⊔ (Expression (“,” ⊔ “●”) MoreExpressionList <aType>))
                                     :MoreExpressionList <aType> |

7 An overloaded procedure is one that takes different actions depending on the
types of its arguments.

8 Note the Symbols box provided by an Integrated Development Environment
(IDE) above to make it easier to construct a program by selecting symbols
from a context sensitive picker. Also an IDE can automatically provide
syntax completion alternatives Analogous to ctrl+space in Eclipse, etc..

9 (“[” ArgumentTypes “]”
  “→” returnType:TypeExpression):ProcedureSignature |
ProcedureSignature ⊆ Expression |
  // signature for a procedure with ArgumentTypes and returnType
( ⊔ MoreArgumentTypes):ArgumentTypes |
(TypeExpression
  ⊔ (TypeExpression “,” MoreArgumentTypes))
                                     :MoreArgumentTypes |

(“Interface” Identifier <Type> “with”
  ProcedureSignatures “.”):ProcedureInterface |
( ⊔ ProcedureSignature
  ( ⊔ MoreProcedureSignatures)):ProcedureSignatures |

10 (“[” ArgumentDeclarations “]” ( ⊔ (“:” Type <aType> ))
  ( ⊔ (“sponsor” Identifier <Sponsor> )))
  “→” Expression <aType>):Procedure |
Procedure ⊆ Expression |
  // Procedure with ArgumentDeclarations that returns
  // Expression of type returnType.
( ⊔ MoreDeclarations):ArgumentDeclarations |
(SimpleDeclaration ( ⊔ (“,” MoreKeywordDeclarations)))
  ⊔ (SimpleDeclaration “,” MoreDeclarations))
                                     :MoreDeclarations |
  // Comma is used to separate declarations.
(((Identifier
  ⊔ (Identifier “:” Expression <Type> ))
  ( ⊔ “default” Expression)):SimpleDeclaration |
(KeywordArgumentDeclaration
  ⊔ (KeywordDeclaration “,” MoreKeywordDeclarations))
                                     :MoreKeywordDeclarations |
(Keyword “⊔” SimpleDeclaration)):KeywordDeclaration |

```

¹⁹ (**"Actor"** ConstructorDeclaration ActorBody):Expression |
 // The above expression creates an Actor with
 // declarations for variables and message handlers
 (⊔ ("extends" ConstructorList))))
 (⊔ "management" Expression <[aType]→Manager>)
 NamedDeclaration
 MessageHandlers
 InterfaceImplementations):ActorBody |
 (Identifier "[" ArgumentDeclarations "]")
 :ConstructorDeclaration |
 (Constructor (⊔ " "))
 ⊔ (Constructor "," MoreConstructors " ")):ConstructorList |
 (Constructor
 ⊔ (Constructor "," MoreConstructors)):MoreConstructors |
 (ActorQueues NamesDeclarations):NamedDeclaration |
 (⊔ (MoreNameDeclarations " ")):NamesDeclarations |
 (NameDeclaration
 ⊔ (NameDeclaration
 "," MoreNamesDeclarations)):MoreNameDeclarations |
 (Identifier
 (⊔ (":" Type <aType>))
 "←" Expression <aType>):IdentifierDeclaration |
 IdentifierDeclaration ⊆ NameDeclaration |
 (Variable (⊔ (":" Type <aType>))
 "!=" Expression <aType> InstanceVariableQualifications)
 :VariableDeclaration |
 VariableDeclaration ⊆ NameDeclaration |
 Variable ⊆ Word |
 InstanceVariableQualifications ⊆ InstanceQualifications |
 (⊔ InstanceVariableQualification
 ⊔ (InstanceVariableQualification
 InstanceVariableQualifications)
 :InstanceVariableQualifications |
 "nonpersistent" ⊆ InstanceVariableQualification |
 // A nonpersistent variable must be of type Nullable<aType>,
 // and can be nulled out before a message is received
 ("queues" QueueNames " "):ActorQueues |
 (QueueName ⊔ (QueueName "," QueueNames)):QueueNames |
 QueueName ⊆ Word |
 QueueName ⊆ Expression <Queue> |
 ("Void"):Expression |

```

(InterfaceImplementation
  ( ⊔ MoreInterfaceImplementations ))
                                     :InterfaceImplementations !
(“also” InterfaceImplementation
  ( ⊔ MoreInterfaceImplementations ))
                                     :MoreInterfaceImplementations !
(( ⊔ “partially”)
  (“implements” ⊔ “reimplements”)
  ( ⊔ “exportable”) Type <aType> “using”
  (MessageHandlers “$”) ⊔ UniversalMessageHandler )
                                     :InterfaceImplementation <aType> !
(MessagePattern
  ( ⊔ (“:” Type ))
  ( ⊔ (“sponsor” Identifier <Sponsor> ))
  “→” ContinuationList <aType> )
                                     :UniversalMessageHandler <aType> !
( ⊔ MoreMessageHandlers ): MessageHandlers !
(MessageHandler
  ⊔ (MessageHandler “¶” MoreMessageHandlers ))
                                     :MoreMessageHandlers !
  // The message handler separator is ¶.
(MessageName “[” ArgumentDeclarations “]”
  ( ⊔ (“:” returnType:Type <aType> )
  ( ⊔ (“sponsor” Identifier <Sponsor> ))
  “→” ContinuationList <aType> ): MessageHandler !
  // For a message with MessageName with arguments,
  // the response is Continuation
(Expression <aType>
  “afterward” Afterward ): Continuation <aType> !
  // Return Expression and afterward perform
  // MoreVariableAssignments
(VariableAssignment
  ⊔ (VariableAssignment
    “,” MoreVariableAssignments “.”))
                                     :VariableAssignments !
(VariableAssignment
  ⊔ (VariableAssignment
    “,” MoreVariableAssignments ))
                                     :MoreVariableAssignments !
(Variable “:=” Expression <aType> ): VariableAssignment <aType> !

```

²⁰ (“**Ⓜ**” anExpression:Expression <aType>
 ((**Ⓜ** (“**sponsor**” Expression <Sponsor>|)):Expression <aType>|
 // Execute anExpression concurrently and respond with the outcome.
 // In every case, anExpression must complete before execution leaves
 // the lexical scope in which it appears.

²¹ (“**Do**” MoreExpressionList
 Continuation <aType> “. ”):Continuation <aType>|
 (Antecedent **Ⓜ** (Antecedent (“,” **Ⓜ** “**●**”) MoreAntecedents))
 :MoreAntecedents|
 Expression **⊆** Antecedent|
 StructureAssignment **⊆** Antecedent|
 ArrayAssignment **⊆** Antecedent|

²² The ability to extend implementation is important because it helps to avoid code duplication.

²³ cf. [Crahen 2002, Amborn 2004, Miller, et. al. 2011]

²⁴ equivalent to the following:
myBalance **⊆** SimpleAccount := **myBalance** **⊆** SimpleAccount – anAmount

²⁵ ignoring exceptions in this way is *not* a good practice

²⁶ (“**Enqueue**” QueueExpression (**Ⓜ** “**after**” Preparation) “**●**”
 Continuation <aType>))) :Continuation <aType>|
 /*
 1. Perform Preparation
 2. Enqueue activity in QueueExpression
 3. Leave the cheese
 4. When the cheese is re-entered perform Continuation . */

(“**■**” Message <aType>):Expression <aType>|
 Delegate message to this Actor.

Cases can be continuations:
 (test:Expression “**◆**”
 ContinuationCases <patternType, aType> “?”)
 :Continuation <aType>|
 (ContinuationCase <patternType, aType>
Ⓜ (ContinuationCase <patternType, aType>
 “**Ⓜ**” MoreContinuationCases <patternType, aType>)))
 ContinuationElseCases)
 :ContinuationCases <patternType, aType>|
 (ContinuationCase <patternType, aType>
Ⓜ (ContinuationCase <patternType, aType>
 “**Ⓜ**” MoreContinuationCases <patternType, aType>)))
 :MoreContinuationCases <patternType, aType>|
 (Pattern <patternType> “:” ContinuationList <patternType, aType>)
 :ContinuationCase < patternType, aType>|
 (**Ⓜ**
 MoreContinuationElseCases <patternType, aType>)
 :ContinuationElseCases <patternType, aType>|

```

((ContinuationElseCase <patternType, aType>
  ⊔ ((ContinuationElseCase <patternType, aType>
    “⊙” MoreContinuationElseCases <patternType, aType>)))
  :MoreContinuationElseCases <patternType, aType>⊥
((“else” “:” ContinuationList <aType>))
  ⊔ (“else” Pattern <patternType> “:”
    ContinuationList <patternType, aType>)))
  :ContinuationElseCase <patternType, aType>⊥
(((Continuation ( ⊔ “.” )))
  ⊔ ((Expression (“,” ⊔ “●”) MoreContinuationList )))
  :ContinuationList ⊥
(((Continuation “.”) ⊔ ((Expression “,” MoreContinuationList )))
  : MoreContinuationList ⊥

```

²⁷ Swiss cheese was called “serializers” in the literature.

²⁸ ReadersWriterConstraintMonitor defined below monitors a resource and throws an exception if it detects that ReadersWriter constraint is violated, e.g., for a resource r using the above scheduler:

ReadingPriority[ReadersWriterConstraintMonitor[r]].

Actor ReadersWriterConstraintMonitor[theResource:ReadersWriter]

writing := False,

numberReading: (Integer thatIs ≥ 0) := 0,

implements ReadersWriter using

read[query] →

Preconditions \neg writing.

Hole theResource.read[query]

after numberReading++

afterward numberReading--

write[update] →

Preconditions numberReading = 0, \neg writing.

Hole theResource.write[update]

after writing:=True

afterward writing := False

²⁹ A downside of this policy is that readers may not get the most recent information.

³⁰ A downside of this policy is that writing and reading may be delayed because of lack of concurrency among readers.

```

31 ("Enqueue" QueueExpression
  ( ( ( "backout" ExpressionList )
    ( ( "after" Preparation ) "●"
      Continuation <aType> ))) :Continuation <aType> |
/*
  1. Perform Preparation
  2. Enqueue activity in QueueExpression
  3. Leave the cheese
  4. When the cheese is re-entered perform Continuation. */
(("■" Message <aType>):Expression <aType> |
// Delegate message to this Actor.
Cases can be continuations:
(test:Expression <patternType> "◆"
  ContinuationCases <patternType, aType> "⊙")
  :Continuation <aType> |
(ContinuationCase <patternType, aType>
  ( MoreContinuationCases <patternType, aType> )
  :ContinuationCases <patternType, aType> |
(ContinuationCase <patternType, aType> (
  (ContinuationCase <patternType, aType>
    "⊙" MoreContinuationCases <patternType, aType> )
  ( ContinuationElseCases <patternType, aType> )
  :MoreContinuationCases <patternType, aType> |
( ( ContinuationElseCase <patternType, aType>
  ( (ContinuationElseCase <patternType, aType>
    "⊙" MoreContinuationElseCases <patternType, aType> )
  :ContinuationElseCases <patternType, aType> |
(ContinuationElseCase <patternType, aType>
  ( (ContinuationElseCase <patternType, aType>
    "⊙" MoreContinuationElseCases <patternType, aType> )
  :MoreContinuationElseCases <patternType, aType> |
(("else" "⊙" ContinuationList <aType> )
  ( ("else" Pattern <patternType> "⊙" ContinuationList <aType> ) )
  :ContinuationElseCase <patternType, aType> |
// The else case is executed only if the patterns before
// the else case do not match the value of test.
(Pattern <patternType> "⊙" ContinuationList <aType> )
  :ContinuationCase <patternType, aType> |

```

³² Precondition that is present for inconsistency robustness.

³³ ++ is postfix increment

³⁴ -- is postfix decrement

³⁵ Precondition that is present for inconsistency robustness.

³⁶ The following are allowed in the cheese for a response to message affecting the next message:

```
((Expression <aType>
  ( ⊔ ("permit" aQueue:Expression)))
  ( ⊔ ("afterward" Afterward))):Continuation <aType> |
/* If there are activities in aQueue, then the one of them gets the
cheese next and also perform Afterward, then leave the cheese
and return the value of Expression. */
```

The following can be used temporarily leave the cheese:

```
("Hole" Expression <aType>):Continuation <aType> |
/*
  1. Leave the cheese
  2. The response is the result of evaluating Expression */
```

```
("Hole" Expression <aType>
  ( ⊔ ("after" Preparation))):Continuation <aType> |
/*
  1. Carry out Preparation
  2. Leave the cheese
  3. The result is the result of evaluating Expression */
```

```
("Hole" Expression <aType>
  ( ⊔ ("after" Preparation)))
  ( ⊔ ("afterward" Afterward)):Continuation <aType> |
/*
  1. Carry out Preparation
  2. Leave the cheese
  3. Evaluate Expression
  4. When a response is received, reacquire the cheese,
carry out Afterward and the result is the result of
evaluating Expression */
```

```

("Hole" Expression<aAnotherType>
  ( ⊔ ("after" Preparation)))
  ( ⊔ ("returned"
      normal:ContinuationCases<aAnotherType, aType> "?")))
  ( ⊔ ("threw"
      exceptional:ContinuationCases<aAnotherType, aType> "?")))
                                     :Continuation<aType>
/*
1. Carry out Preparation
2. Leave the cheese
3. Evaluate Expression
4. When a response is received, reacquire the cheese
   • If Expression returns, continue using the returned
     Actor with normal.
   • If Expression throws an exception, continue using the
     exception with exceptional. */

```

```

37 (Identifier<Type>
  "<" ParametersDeclarations ">"
  "≡" ExpressionList ):ParameterizedDefinition |
ParameterizedDefinition ⊆ Definition |
// Parameterize definition with ParametersDeclarations |
( ⊔ MoreParameterDeclarations ):ParametersDeclarations |
(ParameterDeclaration
  ⊔ (ParameterDeclaration
    "," MoreParameterDeclarations))
                                     :MoreParameterDeclarations |
(Identifier<Type> ( ⊔ Qualifier )):ParameterDeclaration |
( ⊔ ("extends" Identifier<Type> )):TypeQualifier |
(Identifier<Type> "<" Parameters ">"):TypeExpression |
(Identifier<Type>
  ⊔ ( ⊔ (Identifier<Type> "," Parameters ))):Parameters |

```

```

38 ("Discrimination" Identifier<Type>
  MoreTypeDiscriminations "." ):Expression<Type> |
(Identifier<Type>
  ⊔ (Identifier<Type> "," MoreTypeDiscriminations))
                                     :MoreTypeDiscriminations |
(Expression<aDiscriminationType> "Δ" Type<aType>)
                                     :Expression<aType> |
// Discriminate to have the type Type<aType> if possible.
// Otherwise, an exception is thrown.

```

```

(Pattern<aDiscriminationType> "Δ" Type<aType>)

```

```

:Pattern <aType> |
    // If matching Actor is a discrimination that can be discriminated
    // then Pattern must match the discriminate.
    ("∇" Type <aType>):Pattern <aType> |
        // Matching Actor must be discrimination that can be
        // discriminated as aType
39 (Identifier <aType> "[" Arguments "]"):Expression <aType> |
    (Identifier <aType> "[" Patterns "]"):Pattern <aType> |
40 ("Structure" Identifier <Type> "[" FieldDeclarations "]")
    ( ⊔ ("extends" ConstructorList))
    NamedDeclaration
    MessageHandlers
    MoreInterfaceImplementations):Definition |
    // Structure definition with StructureImplementation
    ( ⊔ MoreFieldDeclarations):FieldDeclarations |
    ((SimpleFieldDeclaration
        ( ⊔ ("," MoreNamedFieldDeclarations)))
        ⊔ (SimpleFieldDeclaration
            "," MoreFieldDeclarations)):MoreFieldDeclarations |
    ((Identifier
        ⊔ (Identifier ":" TypeExpression))
        ( ⊔ "default" Expression)):SimpleFieldDeclaration |
    (NamedFieldDeclaration
        ⊔ (NamedFieldDeclaration
            "," MoreNamedFieldDeclarations))
        :MoreNamedFieldDeclarations |
    (FieldName
        ("⊔" ⊔ ":" ⊔) SimpleFieldDeclaration))
        :NamedFieldDeclaration |
    FieldName ⊆ QualifiedName |
        // ":" is used for assignable fields.
    (( ⊔ Identifier) ActorBody):StructureImplementation |
    (Expression "[" FieldName "]" ):FieldSelector |
        // FieldName of Expression which must be a structure
    FieldSelector ⊆ Expression |
    (StructureName "[" FieldExpressions "]" ):StructureExpression |
    StructureExpression ⊆ Expression |
    ( ⊔ MoreFieldExpressions):FieldExpressions |
    ((SimpleFieldExpression( ⊔ ("," MoreNamedFieldExpressions)))
        ⊔ (SimpleFieldExpression
            "," MoreFieldExpressions)):MoreFieldExpressions |
    (NamedFieldExpression

```

```

    ⊔ ( NamedFieldExpression
        “,” MoreNamedFieldExpressions))
                                     :MoreNamedFieldExpressions |
(FieldName
    (“⊔” ⊔ “:⊔”) SimpleFieldExpression))
                                     :NamedFieldExpression |
(StructureName “[ FieldPatterns ]”):StructurePattern |
StructurePattern ⊆ Pattern |
( ⊔ MoreFieldPatterns):FieldPatterns |
((SimpleFieldPattern( ⊔ (“,” MoreNamedFieldPatterns))))
    ⊔ ( SimpleFieldPattern “,” MoreFieldPatterns))
                                     :MoreFieldPatterns |

(NamedFieldPattern
    ⊔ ( NamedFieldPattern
        “,” MoreNamedFieldPatterns))
                                     :MoreNamedFieldPatterns |
(FieldName (“⊔” ⊔ “:⊔”) SimpleFieldExpression))
                                     :NamedFieldPattern |
41 (“Try” anExpression:Expression <aType>
    “catch” ExpressionCases <Exception, aType> “[?]”)
                                     :Expression <aType> |

/*
    • If anExpression throws an exception that matches the pattern
      of a case, then the value of TryExpression is the value
      computed by ExpressionCases
    • If anExpression doesn't throw an exception, then then the
      value of TryExpression is the value computed by
      anExpression. /*

(“Try” anExpression:Expression <aType>
    “catch” ContinuationCases <Exception, aType> “[?]”)
                                     :Continuation <aType> |

/*
    • If anExpression throws an exception that matches the pattern of
      a case, then the response of TryContinuation is the
      response computed by the expression of the case.
    • If aContinuation doesn't throw an exception, then then the
      response of TryExpression is the response computed by
      anExpression. */

```

```

(“Try” anExpression:Expression <aType>
    86

```

```

“cleanup” cleanup:Expression<aType>):Expression<aType>|
*/
    • If anExpression throws an exception, then the value of
      TryExpression is the value computed by cleanup.
    • If anExpression doesn’t throw an exception, then then the
      value of TryExpression is the value computed by
      anExpression. */
42 (“Preconditions” test:ExpressionList ExpressionList):Expression |
    // Each of expressions in test must evaluate to True or
    // an exception is thrown
    (“Preconditions” ExpressionList ContinuationList):Continuation |
    // Each of expressions in ExpressionList must evaluate to True or
    // an exception is thrown
    (value:Expression <aType>
    “postcondition” pre:Expression <[aType]→Boolean>
    :Expression <aType>|
    // The expression pre must evaluate to True when sent value
    // or an exception is thrown
43 ° is a reserved postfix operator for degrees of angle
44 i.e., i*i=-1 where i is the imaginary number Cartesian[0, 1]
45 (“[” ComponentExpressions “]”):Expression <List>|
    // An ordered list with elements ComponentExpressions
    ( [ MoreComponentExpressions ):ComponentExpressions |
    ((( [ “V” ) Expression )
    [ (( [ “V” ) Expression
    “,” MoreComponentExpressions )))
    :MoreComponentExpressions |
    (“[” TypeExpressions “]”):TypeExpression |
    ( [ MoreTypeExpressions ):TypeExpressions |
    (TypeExpression [ (TypeExpression “,” MoreTypeExpressions )))
    :MoreTypeExpressions |
46 (“_”):UnderscorePattern |
    UnderscorePattern ⊆ Pattern |
    Identifier ⊆ Pattern |
    (Pattern “suchThat” Expression):SuchThat |
    SuchThat ⊆ Pattern |
    (Pattern “thatIs” Expression):ThatIs |
    ThatIs ⊆ Pattern |
    (“$$” Expression <Type>):Pattern <Type> |
    (“[” ComponentPatterns “]”):Pattern <List> |
    // A pattern that matches a list whose elements match
    // ComponentPatterns

    ( [ MoreComponentPatterns ):ComponentPatterns |

```

(Pattern

```

    ⊔ ( "V" Pattern )
    ⊔ ( Pattern " ," MoreComponentPatterns ))
                                     :MoreComponentPatterns |
47 ( "{ ComponentExpressions " } ): Expression <Set> |
    // A set of Actors without duplicates
    ( "{ ComponentPatterns " } ): Pattern <Set> |
48 ( "{ ComponentExpressions " } ): Expression <Multiset> |
    // A multiset of the Actors with possible duplicates
    ( "{ ComponentPatterns " } ): Pattern <Multiset> |
49 Optimization of this program is facilitated because:
    • The records are determinate because their type is
      Set<ContactRecord>
    • All of the operators return determinate results
    • The operators are annotated as determinate
50 ( "Map" "{ ComponentExpressions " } ): Expression <Map> |
51 It is possible to define a procedure that will produce a "bottomless" future.
    For example, f.[ ]:Future<aType> ≡ Future f.[ ] |
52 the examples using ⊕ can be slightly more efficient as written
53 ( Postpone Expression <aType> | ): Expression <Future<aType>> |
    // postpone execution of the expression until the value is needed.
54 ( "Future" aValue:Expression <aType>
    ( ⊔ ( "sponsor" Expression <Sponsor> )))
                                     :Expression <Future<aType>> |
    // A future for aValue.
    ( "↓" Expression Future <aType>> ): Expression <aType>> |
    // Resolve a future
55 ( LoopName:Identifier " ." "[ Initializers " ]
    ( ⊔ ( " ." Return Type:aType ))
      "≡" Expression <aType> ): ExpressionList <aType> |
    ( ⊔ MoreInitializers ): Initializers |

    (Initializer ⊔ (Initializer " ," MoreInitializers))
                                     :MoreInitializers |
    (Identifier ( ⊔ ( " ." TypeExpression )) "←" Expression ): Initializer |
56 The implementation below requires careful optimization.
57 ( "String" "[ ComponentExpressions " ] ): Expression <String> |
    ( "String" "[ ComponentPatterns " ] ): Pattern <String> |
58 ( recipient:Expression <recipientType>
    " ." message:MessageExpression <recipientType> ): Expression |
    // Send recipient the message
59 /* A Postpone expression does not begin execution of Expression1 until a
    request is received. Illustration:
    IntegersBeginningWith.[n:Integer]:<FutureList<Integer> ≡

```

[n, vPostpone IntegersBeginningWith_n[n+1]]

Note: A Postpone expression can limit performance by preventing concurrency */

- ⁶⁰ (((" MoreGrammeters ")):Grammar |
(((" Grammar " " Grammar ")):Grammar |
(ReservedWord (" StartsWithIdentifier))):StartsWithReserved |
StartsWithReserved \sqsubseteq MoreGrammeters |
(Identifier (" StartsWithReserved))):StartsWithIdentifier |
StartsWithIdentifier \sqsubseteq MoreGrammeters |
(\" Word \") :ReservedWord |
// The use of \ escapes the next character in a string so
// that \" has just one character that is \.
(Grammar ":" GrammarIdentifier " "):Judgment |
(Identifier<Grammar> " " Identifier<Grammar> " "):Judgment |
- ⁶¹ The implementation below can be highly inefficient.
- ⁶² ("Atomic" aLocation:Expression
"compare" comparison:Expression
"update" update:Expression " " " \diamond "
"updated" "s" compareIdentical:ContinuationList<aType> " " " \oplus "
"notUpdated" "s" compareNotIdentical:ContinuationList<aType> " " " \otimes ")
:Continuation<aType> |
/* Atomically compare the contents of aLocation with the value of
comparison. If identical, update the contents of aLocation with the
value of update and execute compareIdentical.
- ⁶³ (Identifier " " Qualifier):QualifiedName |
QualifiedName \sqsubseteq Expression |
Identifier \sqsubseteq QualifiedName |
(Identifier (Identifier " " Qualifier)):Qualifier |
- ⁶⁴ ("Enumeration" Identifier<Type>
MoreEnumerationNames ". "):Definition |
(EnumerationName
" (EnumerationName
" " MoreEnumerationNames))
:MoreEnumerationNames |
EnumerationName \sqsubseteq Word |
- ⁶⁵ Declarations provide version number, encoding, schemas, etc.
- ⁶⁶ If a customer is sent more than one response (i.e., return or throw message) then it will throw an exception to the sender of the response.
- ⁶⁷ (recipient:Expression
" " MessageName " [Arguments]):Expression<Void> |

/* recipient is sent one-way message with *MessageName* and *Arguments*. Note that *Expression* $\langle \ominus \rangle$ cannot be used to produce a value. */

```

68 ((MessageName [" ArgumentDeclarations "]
    ( ( ( "sponsor" Identifier <Sponsor> ) ) ) ) )
    "→" ContinuationList <⊖> : MessageHandler !
    /* one-way message handler implementation with
       ArgumentDeclarations that has a one-way continuation
       that returns nothing */
    ( "⊖" ( ( ( "permit" aQueue: Expression ) )
        ( ( ( "afterward" Assignments ) ) ) : Continuation <"⊖"> !

```

⁶⁹ Hoare[1962]. The implementation below is adapted from Wikipedia.

⁷⁰ // Move Actor at pivotIndex to end

⁷¹ /* Consider a dialect of Lisp which has a simple conditional expression of the following form:

```
((("if" test: Expression then: Expression else: Expression))
```

which returns the value of then if test evaluates to **True** and otherwise returns the value of else.

The definition of Eval in terms of itself might include something like the following [McCarthy, Abrahams, Edwards, Hart, and Levin 1962]:

```
(Eval expression environment) ≡
    // Eval of expression using environment defined to be
    (if (Numberp expression) // if expression is a number then
        expression // return expression else
        (if ((Equal (First expression) (Quote if))
            // if First of expression is "if" then
            (if (Eval (First (Rest expression)) environment)
                // if Eval of First of Rest of expression is True then
                (Eval (First (Rest (Rest expression)) environment)
                    // return Eval of First of Rest of Rest of expression else
                (Eval (First (Rest (Rest (Rest expression)) environment))
                    // return Eval of First of Rest of Rest of Rest of expression
            ...))

```

The above definition of Eval is notable in that the definition makes use of the conditional expressions using **if** expressions in defining how to evaluate an **if** expression! */

⁷² The implementation **CheeseQ** uses activities to implement its queue where for type **Activity** the following holds:

```

Structure Activity [previous : ⊡ Nullable <Activity>,
    // if null then head of queue else,
    // pointer to backwards list to head
    nextHint : ⊡ Nullable <Activity>] !
    // if non-null then pointer to next
    // activity to get cheese after this one

```

⁷³ If non-null points to head with current holder of cheese

⁷⁴ If non-null, pointer to backwards list ending with head that holds cheese

⁷⁵ Interface **CheeseQ** with **enter[]** → **Void**,

```

    leave[ ] ↪ Void. !
76 // enter message received running myActivity
77 /* this cheese queue is not empty because myActivity is at the head of
    the queue */
78 Interface SubCheeseQ with head[ ] ↪ Activity!
79 Interface InternalQ with enqueueAndLeave[ ] ↪ Void,
    enqueueAndDequeue[InternalQ] ↪ Activity,
    dequeue[ ] ↪ Activity,
    isEmpty[ ] ↪ Boolean. !
80 Interface SubInternalQ with add[Activity] ↪ Void,
    remove[ ] ↪ Activity. !
81 [Church 1932; McCarthy 1963; Hewitt 1969, 1971, 2010; Milner 1972,
    Hayes 1973; Kowalski 1973]. Note that this definition of Logic Programs
    does not follow the proposal in [Kowalski 1973, 2011] that Logic Programs
    be restricted only to clause-syntax programs.
82 A ground-complete predicate is one for which all instances in which the
    predicate holds are explicitly manifest, i.e. instances can be generated using
    patterns. See [Ross and Sagiv 1992, Eisner and Filardo 2011].
83 Execution can proceed differently depending on how sets fit into computer
    storage units.
84 following expression is executed concurrently
85 Used in type specifications for interfaces.
86 Used in message handlers.
87 Used to bind identifiers in Let.
88 Three equal signs because two equal signs have a meaning in Java
89 Used in patterns.
90 Used in structures.
91 Used in one-way message passing.

```