The *Honeysuckle* Programming Language

A (draft) manual (HPL-2010)

Ian East January 2010

©2006-9 Dr. Ian Robert East *all rights reserved*

This document may not be reproduced, either in part or in its entirety, by any means, including manual, mechanical, or electronic, without the express *written* consent of the author.

Caution! This document is a *draft* only. Content is both incomplete and by no means fully verified. It undoubtedly contains errors.

It is being made public only for purposes of debate and development. Please report any error or constructive criticism to the author.

It will be updated periodically until stability has been achieved.

Simplicity is the ultimate sophistication.

attr. Leonardo da Vinci (1452-1519)

Purely applicative languages are poorly applicable.

Alan Perlis (1922-1990)

Changes from previous (January 2009) edition

1 Applicable formal conditions and design rules have been restored (#1.4).

Design rules have been brought together therein and are no longer distributed through the main text.

They have been attributed a new enumeration, and the wording of one changed, for the sake of clarity.

2 Change of keyword: IDLE now QUIET within an alternation.

3 Minor corrections throughout.

4 References tidied up and reduced to those cited.

Changes anticipated in next edition (January 2011)

1 ADT class defined, to include:
- applicable operators (value definitions)
- invariants
- genericity.

2 Infix operator definition.

3 HVDL:
- colouration defined, appropriate for either navy blue or black background
- modularity indicated by open box or ovoid (terminal or interface)
- visual syntax for repetitive structure in network and interface.
- all figures to be coloured correctly.

©2006-10, Dr. Ian Robert East

1 Introduction

1.1 Purpose of this document

Here you will find a summary of every element of the Honeysuckle Programming Language (HPL) and its model for system (and project) abstraction. You will find little explanation or justification. This manual simply provides a connection between the means of system abstraction and the symbols, productions, and semantic rules, of the language. The intention is to provide a readable definition of the language. There are no complete examples herein.

In order to retain readability, only a subset of the semantic rules required are included, where something different about Honeysuckle needs emphasis or clarification. However, all aspects of the language are described. It is merely left to the aspiring compiler writer to translate every constraint into a precise rule whose verification can be automated.

Syntax, on the other hand is complete and summarized in Appendix A.

Appendix B summarizes the Honeysuckle Core Language (HCL) – a subset of HPL.

1.2 Aims of the Honeysuckle project

1.2.1 For whom

Honeysuckle is primarily intended to allow the fresh (hitherto uncorrupted) *student* of programming to express behaviour intuitively, using a natural model for abstraction that includes concurrency and prioritized alternation between processes.

Honeysuckle may not appeal to those in both the commercial and academic programming communities who revel in obscure or unnecessarily formal notation, or who believe they suffer no obligation to compose programs readable by others.

While it may not prove immediately popular with programmers used to conventional languages, it will hopefully appeal to those to whom they are responsible. Honeysuckle has much to offer the *project manager*, who must account for the poor readability that inevitably results from a stylistic freedom afforded each individual programmer, and who must somehow make components live up to their 'paper' specification.

Neither students nor project managers profit from either stylistic freedom or the lack of a common language with which to both define and implement a system. Honeysuckle quite deliberately sacrifices stylistic liberty on the altar of readability and transparency.

The final customer category is the *engineer*, who would normally expect a method to possess a formal foundation, and to be scalable to systems with any degree of size and complexity. Engineers also expect design to be deliverable prior to, and separate from, any implementation, and to satisfy rules that are proven to exclude pathological behaviour,

1.2.2 To what end

For the student, Honeysuckle provides a vehicle for learning to program *ab initio*, without the usual restriction to purely sequential and proactive behaviour. Because its model for abstraction extends to concurrency and prioritized alternation, it is an appropriate classroom tool for programming reactive behaviour in (for example) embedded systems. Because it possesses a formal foundation and static verification of formal design rules, it is appropriate to modern applications which incorporate high expectations for system integrity at low cost.

The student also benefits from the amicable divorce of formal analysis from practical implementation. While the formal properties of every program remain open to mathematical investigation, it remains unnecessary to conduct an analysis in most applications.

With Honeysuckle, it is possible to separate concerns – theory from practice, design from implementation, *etc.* – which is of the greatest assistance in learning.

For the project manager, Honeysuckle offers the prospect of increasing system integrity while decreasing the time and cost associated with both development and maintenance. To this end, it seeks to integrate design, (static) verification, and implementation, while remaining accessible to engineers with moderate – and thus easily recruited – skills.

For the engineer, a stand-alone design may be expressed, free of any implementation. Each design may be independently verified, before proceeding to implementation. Every admitted implementation will be guaranteed to refine design, and thus retain security against error.

Concurrency and prioritized reactive behaviour incur considerably greater scope for error, against which a predecessor, occam [1], offered little protection. Honeysuckle enforces formal design rules that exclude, at least, the most insidious threat. *Every Honeysuckle program inherently denies the possibility of deadlock.* (A formal proof has been published.)

In short, Honeysuckle allows more liberal and transparent expression of a greater variety of behaviour, with a lower expectation of skill, and with more errors automatically excluded. In other words, it offers cheaper, better, software, faster.

Honeysuckle is also a response to certain inadequacies in its antecedent, occam. While the latter represented unprecedented innovation, greatly simplifying the secure programming of concurrent/reactive systems, it provided poorly for data abstraction and project modularity. It also limited inter-process communication to the copying of value, limiting abstraction, and causing inefficiency in many applications.

In Honeysuckle, one may either send a value or *transfer an object* to a concurrent process, just as one can in Pascal to a procedure (a process in sequence). Objects are thus permitted *mobility* between processes, affording the transparent expression of distributed systems.

While occam offers only processes in its model for abstraction, a typical OOP language offers only objects. Honeysuckle offers complete and *balanced abstraction*. Object and process reside together in harmony. Processes own, and sometimes trade, objects.

©2006-10, Dr. Ian Robert East

1.3 About Honeysuckle

1.3.1 Modularity

Honeysuckle recognizes the need for *both* an adequate model for system abstraction *and* an adequate model for the separated development and reuse of software.

System modularity expresses how the system is reduced into communicating processes and the objects they each own.

Project modularity expresses the reduction of a project into *items* (individual definitions) and *collections* of items, that can be separately developed and reused in other projects.

1.3.2 Model for abstraction

Elements and terminology

A system consists of component processes which may be composed in sequence or parallel. Those in sequence communicate via *objects*, those in parallel communicate via the provision and consumption of *services*.

An object is something on which a *value* may be written.

A service comprises a designated sequence of oriented synchronous *communications*.

Every service is implemented by *ports* at either end. The *interface* of any process is a list of attributed *client* and *server* ports. Every process is specified by its interface.

The term *item* refers to process, object, or service. Any item may be separately defined ('offline'), to allow reuse within a single application or between multiple applications. Where it is used just once, it must be defined at that point ('inline').

Recursion may be used in the definition of any item.

The term *value* corresponds with constant, and *object* with variable, in Pascal. Like Pascal, the *class* of value that may be written on any object cannot vary. Unlike Pascal, class definition may be recursive (hence the number of elements in a compound type may vary) and extends to include applicable operators and invariants.

Values are named and defined simultaneously. Objects are first *named* (declared) and later explicitly created. Between declaration and creation, only the name exists. Each object is ultimately destroyed, either by explicit command or automatically when its *owner* terminates.

Process and object

Every object is owned by exactly one process at any time. Ownership may be transferred between processes composed in parallel (by command) or sequence (by declaration). Objects are thus mobile within a process network.

An object passed to a concurrent process constitutes a *gift* and becomes the responsibility of the recipient. Unlike a value passed, it ceases to be visible within the donor. No further reference is permitted. Unless explicitly returned, destroyed or, passed on, its existence will end when the new owner terminates.

In contrast, an object passed to a process in sequence is a *loan* and is returned upon termination of the recipient. Passing value or object to a sequential component corresponds to that of a 'value' or 'reference' parameter to a Pascal procedure. Like Pascal, each may adopt a local alias. Unlike Pascal, a value may not be changed by the recipient. Also unlike Pascal, a third possibility exists. A *name* may be passed for an object the recipient must create and return.

Classes are "name equivalent", not "structure equivalent". For example, this means integer length and integer time are distinguished. It also implies that types, as well as values and objects, must have the ability to persist between process runs. Otherwise persistent data cannot be interpreted.

Service

A service is simply a sequence of *communications* between two concurrent processes. Its prior definition and agreement constitute a protocol governing their interaction.

Every communication is an element of some service. Every service provided/consumed by any component is either provided/consumed by another or forms part of the system interface.

Each element within a service infers:

- the type of value communicated
- whether value or object (with value) is transferred
- the direction of transfer.

Service is always initiated by the *client*, which then guarantees complete consumption. Following initiation, the *server* makes a similar guarantee regarding provision, although it may delay each element while it consumes some other service.

Miscellaneous features and attributes

To avoid explicit distinction between object and value, and the consequent necessity of introducing pointer or reference notation, Honeysuckle instead makes the intention *implicit* according to the command issued. For example, we SEND a value, but TRANSFER an object.

Inheritance of any *context* by a block (sequential or parallel construction) from its parent is always explicitly declared. As a result, semantic equivalence is established between a block defined in-line and off-line (a 'procedure').

Either value or object may be passed between processes running in parallel. Symmetry is thus established in communication between processes running in sequence and parallel.

'do-while-do' repetition is added to the sequential behaviour programmable in Algol or Pascal. A complete set of 'prime' programs, noted by Maddux [2], thus becomes available.

Recursive data type and *class* may be expressed, along the lines discussed by Hoare [3], without any need for references, pointers, or automated garbage collection.

Dynamic replication allows class definition of objects which vary in size without recourse to recursion. This is intended as support for the efficient representation of text (strings).

Genericity of replicative and recursive classes (data type), with regard to a single component type is also available. One may thus separate the concerns of structure and element.

 ©2006-10, Dr. Ian Robert East

1.4 Prioritized Service Architecture (PSA)

1.4.1 Introduction

PSA forms the formal foundation for Honeysuckle, and is itself founded upon the theory of *Communicating Sequential Processes* (CSP) [4] and the *failures* model for its semantics [5].

A formal definition of service protocol and service network have been published accordingly, along with a proof of deadlock freedom, subject to certain *design rules*, that originated with J. M. R. Martin [6, 7]. These, and the conditions that constrain the definition of service and service network, are summarized here, briefly and informally.

Service and service network conditions are not necessarily verified directly, but may be shown the consequence of syntax and further design rules that apply to the directed graph which represents the prioritized service network declared within a program. Such rules are described in this document once the necessary syntax has been presented.

1.4.2 Service

Two formal conditions permit a definition of service [7]. Informally, these are as follows:

Client Condition A client may initiate service at any time by requesting the first communication, but must then eventually offer each subsequent communication according to the definition of the service concerned. It must continue to offer each communication until granted.

Server Condition A provider must initially offer only the first communication, and must eventually grant each subsequent communication until completion.

The ability of any process to fulfil these conditions depends upon obligations that arise in the provision or consumption of other services. Certain constraints (design rules) must thus be imposed upon the combination of services within any design.

Every component must declare its *interface* in the form of services consumed and services provided. These shall be referred to as *elements* of its interface. The component must eventually offer the first communication of any service provided regardless of the progress of any other service, to guarantee the Server Condition.

The Client Condition is similarly protected by design rules.

Design rules govern the way in which components may be connected, and require verification only within a design, since any implementation is, in turn, required to satisfy that design.

Note that *sequential* service provision is only possible through sequential service composition. Parallel composition must be through a (prioritized) interleaving.

1.4.3 Service network

Servers may be collected together, so that each is a member of exactly one *bunch*. Within any single bunch, service provision is mutually exclusive. From the moment the delivery of any service starts until completion, no other member of its bunch may be offered.

©2006-10, Dr. Ian Robert East

A *dependency* may exist between component server and client connections. Any dependency applicable to a server connection is shared by every member of its bunch, and is transitive.

Each bunch, together with any dependent clients is termed a *service interface component*.

Every *service network component* (SNC) must satisfy four formal conditions [7:

Network Communication Condition

Every communication forms part of some predefined service.

Network Composition Condition

Every service provided/consumed by a component is either consumed/provided by another or forms an element of the system interface.

Network Client Condition

No component may ever refuse everything within any service interface component.

Network Server Condition

A component must offer either all services provided within a service interface component or none.

Together, these ensure that every SNC is 'live', *i.e.* cannot refuse all communication.

The definition of a SNC is closed under parallel composition. Every component forms a valid system and every system a valid component. Designing a system with *service architecture* may be deemed 'compositional' in this sense.

Any SNC whose interface consists of a single server bunch is indivisible.

Any two services within a dependency may proceed either in *sequence* or in parallel, with their communications *interleaved*. This distinction applies either to services across a dependency (one dependent upon the other) or to services upon both of which another depends.

This distinction is enough to permit the denial of resource allocation deadlock (RAD), via design rules that deny insecure combination of chains of dependency.

One further relation between services is necessary to permit adequate expression of *reactive* (event-driven) behaviour. The provision of one service may *pre-empt* another, once initiated by a client. It will then continue until *completion*, perhaps subject to interruption itself, before the original service is resumed. The sequence in which services provided by a single component may be interrupted is referred to as a *prioritization*.

Any system which may be described by a graph expressing service and prioritization is said to possess *prioritized service architecture* (PSA).

1.4.4 Formal design rules and deadlock avoidance

Because Honeysuckle has a formal foundation, it is possible to prove consequences of adherence to certain design rules.

Four design rules must govern PSA to deny deadlock:

Design Rule 1 *circular dependency*

There must be no directed circuit of dependency and pre-emption.

 ©2006-10, Dr. Ian Robert East

Design Rule 2 *crossed dependency*

Any two services which mutually depend upon shared, exclusive, or interleaved, provision must be consumed in sequence, and not (interleaved) in parallel.

Design Rule 3 *dependency in interleaved provision*

When pre-emption and dependency are orthogonally aligned, no pair of feed-forward dependency chains may cross.

Design Rule 4 *feedback within an interleaving of provision*

Any feedback must ultimately depend upon a service provided at higher priority, *i.e.* the *feedback chain* must end at a point higher than its start.
(This requires feedback to be indicated upon provision before consumption.

In any feedback chain, at least one dependency must be interstitial.

Any additional dependency of the lower-priority service must be arranged in sequence, and not interleaved, with the feedback path when fed forward to further interleaved provision, along with the higher-priority path. The order of the sequence is immaterial.

Note that any interstitial element in a chain of dependency makes no difference to Design Rule 2. It also applies when no intervening dependency exists, and provision is direct.

Verification is possible prior to execution and requires the incremental construction of a two-colour prioritized service digraph (PSD), wherein each dependency is aligned along one axis (say, left-to-right) and each pre-emption along another orthogonal (say, bottom-to-top).

These rules guarantee the absence of deadlock, including that due to either concurrency or prioritization, and of all priority conflict (and thus of priority inversion).

1.5 Implementation

1.5.1 Tools

Separate translators are needed for design and implementation elements, plus a tool for the capture of graphical design using the *Honeysuckle Visual Design Language* (HVDL) and its translation into textual form, using the *Honeysuckle Design Language* (HDL).

In summary, the complete Honeysuckle programming language (HPL), incorporating HVDL and HDL, requires for its implementation the following four tools:

–	HSG	Honeysuckle Scanner-Generator	lexical analysis
–	HVDC	Honeysuckle Visual Design: Capture	capture of graphical design translation to (textual) HDL
–	HDVC	Honeysuckle Design: Verify and Compile	verify design against formal rules compile (translate to binary)
–	HPLC	HPL: Compile	compile (incorporating HDVC).

There is an obvious hierarchical interdependence between the above.

It is anticipated that compilation will target an intermediate language, representing a virtual machine capable of executing suitable instructions that efficiently support concurrent and reactive behaviour. One appropriate instruction set is the *Extended Transputer Code* (ETC), developed from that of the Inmos Transputer in the mid 1980s (documented elsewhere).

©2006-10, Dr. Ian Robert East

1.5.2 A Honeysuckle Core Language (HCL)

A complete implementation of Honeysuckle is a somewhat daunting task, and likely to prove expensive. Therefore, an incomplete Honeysuckle is proposed, to be known as the *Honeysuckle Core Language* (HCL). (The pantheon of acronyms is now hopefully complete.)

A programming language is holistic to a great degree, making it difficult to reduce. Reduction is only possible by admitting change. What remains is no longer truly Honeysuckle, but it nonetheless retains its philosophy and approach.

To avoid repetition, and the possibility of documentation falling "out of sync", HCL is described alongside HPL in one common manual (this one). A '‡' to the left of a production indicates syntactic variation between core and complete language. (It may indicate the loss of the entire production.) Appendix A offers a summary of HPL, Appendix B of HCL.

An implementation of HPLC need accept only syntax described under the label CORE. Unless it implements the complete language, it should be referred to as HPLC-CORE.

The following features are absent in HCL:

- the Honeysuckle Visual Design Language (HVDL)
- project-modularity (collection definition)
- composition data type (object class) definition (records and arrays)
- recursive data type (object class) definition
- generic data structure (independent of element type)
- abstract data types (encapsulation of applicable routines and class invariant)
- conditional, repetitive, or reursive, structure within service definition
- shared, synchronized, prioritized, and distributed (concurrent), service provision
- chaining, feedback, or replicated alternation, of service
- network parameters
- real-time primitives (MARK and WAIT)
- timed selection criteria (AFTER and BEFORE)
- function (VALUE OF) definition
- prefix expressions
- vector assignment.

The following core features are retained in HCL:

- system-modularity (process, service, and class, definition)
- prioritized alternation (WHEN construct, and in NETWORK declaration)
- mobility of objects between processes (transfer of ownership)
- deadlock-freedom (static verification of formal design rules).

In other words, HCL affords only simple (enumerated and 'built-in') data type, and simple sequential service, definition. Design offers no "bulk synchrony" and no concurrent service provision. However, the model for abstraction retains secure concurrent, and prioritized reactive, behaviour, along with objects that are mobile between processes.

Core Honeysuckle will only ever be applicable to simple systems. It will not scale to larger ones because of the loss of project-modularity and the opportunity for visual design.

 ©2006-10, Dr. Ian Robert East

2 Notation and annotation

2.1 Notation

2.1.1 Character set

Program character set

Literal characters appearing within the program text, and those appearing at program input or output, are from here onward referred to as *processed text*. This is considered distinct from *program text*. Each is ascribed a different legitimate character set. That of processed text is a superset of (contains) that of program text.

A Honeysuckle program, with the exception of literal characters and strings, is inscribed using only the following characters:

–	digits	`0–9`	– symbols	`# \`
–	letters	`a–z A–Z`	– operators	`+ − × ÷ \ / ^ ∧ ∨ ¬`
–	brackets	`( ) [ ] { }`	– relations	`= ≠ < > ≤ ≥`
–	punctuation	`: ' " , . ?`	– format	space NBS SNL

NBS refers to a "non-breaking space", which is used as an option to bind multiple words together to form a name. SNL denotes a command to "start new line". (It is a common mistake to regard a control character (sequence) at the end of a line as a mark of its end. A mark is required to *delimit* successive lines, and is merely one possible indication of the end of one.)

Both SNL and 'space' characters are elements of Honeysuckle syntax. They are *not* ignored as mere "white space" as they are in other languages.

While Honeysuckle is independent of any particular character encoding scheme, it is expected that this will be the 16-bit *Unicode Translation Format*, UTF-16. Anything more is unnecessary; anything less is inadequate.

There follows the 16-bit *Unicode Translation Format*, UTF-16 encoding:

	Unicode		Unicode		Unicode		Unicode
×	#00D7	NBS	#00A0	¬	#00AC	÷	#00F7
∧	#2227	∨	#2228				
≠	#2260	≤	#2264	≥	#2265		

The remainder fall within the ASCII 7-bit, subsumed by UTF-8 and UTF-16.

No commitment is made here to any aspect of the binary representation of a Honeysuckle program. It may be that a string, after translation, is terminated by an ASCII NUL, but that need not form part of a language definition.

Nor does it prove necessary to incorporate the detection of the end of the program text file. An end-of-file (EOF) is never required as the terminator of any lexeme (see below).

Processed character set

Apart from character and string literals, any programming language can be defined independently of character encoding, which would otherwise remain an implementation issue only. However, if the *processed character set* (PCS) is allowed to differ between implementations, then programs cannot be guaranteed portable – a character referred to in one implementation may not be available in a second. For this reason, Honeysuckle defines this set also.

While it is desirable to offer the largest character set possible, for processed text, some consideration must also be given to implementation. Practicability demands the adoption of a set defined along with a standard encoding, and that a large set is divided into a small number of subsets. Performance and ease of implementation also benefits from a fixed-length encoding.

For these reasons, every Honeysuckle program is expected to be encoded using the 16-bit *Unicode Translation Format*, UTF-16. Refer to the Unicode Standard [8]. The Latin character set of the ASCII remains, and standard mathematical symbols can represent mathematical operators. Common symbols, such as ©, £, and €, are also available.

One departure from UTF-16 is that character codes are regarded as of fixed length, whereas the Unicode standard allows for 'surrogates' – additional 16-bit words that yield a 32-bit code. In the vocabulary of Unicode, only the "basic multilingual plane" (BMP) is admitted. *Care must be taken not to allow entry of any characters beyond the BMP.*

Note that Unicode defines characters, not glyphs. Choice of font is left to the Honeysuckle Programming Environment (HPE).

Byte-ordering may follow that of the system, and does not interact with language definition.

Limited use of the set defined by UTF-16 is made for programming. An extended domain is available for processed text, though restricted a little to ease lexical analysis. No control character is included. Any detected will be identified as a *gremlin*. However, reference can be made to *any* character indirectly via an *escape sequence*.

The PCS reduces to those characters whose UTF-16 encoding lies between:

– `#0020–007E` or `#00A0–0D7FF`

Many codes therein remain as yet undesignated. There is no obligation to detect these.

Three distinct categories of gremlin should be reported by the lexical analyst (scanner):

–	an unsupported control code	in	`#0000–001F`	`#007F–009F`
–	an unsupported character	in	`#F900–FDCF`	`#FDF0–FDFD`
–	no character	in	`#D800–DFFF`	`#E000–F8FF`
			`#FDD0–FDEF`	`#FFFE–FFFF` .

An escape sequence begins with either '\' or '#'. A backslash is followed by a single character from the set {n _ # \ ' "}. A '#' is followed by four hexadecimal digits – 0…9, A…F. (Note the use of capital letters to denote digits ten to fifteen.)

The following table summarizes:

 ©2006-10, Dr. Ian Robert East

`\n`	line-feed (new line) NL	`\#`	#
`_`	non-breaking space NBS	`\\`	\
`\'`	'	`\"`	"
`#hhhh`	character whose encoding is $hhhh_{16}$ (h = hex digit)		

There is no restriction on the four hex digits following a '#' ('hash'), allowing codes:

– `#0000–FFFF`

While this does allow access to the full *Universal Character Set* (UCS), including members with 32-bit encoding (via two codes in sequence), it also allows both undesignated and 'illegal' codes. The programmer is responsible for any undesired behaviour that results.

The following equivalences apply between escape sequences:

– start new line SNL `#000A` `\n`
– non-breaking space NBS `#00A0` `_`

Character classes

A single class defines the set of admissible printing characters, via their UTF-16 encoding:

any ⟶ #0020–#007E | #00A0–#D7FF

In addition, the following (disjoint) character classes will be needed:

space ⟶ #0020
SNL ⟶ #000A
NBS ⟶ #00A0
digit ⟶ `0–9`
letter ⟶ `a–z | A–Z`
quote ⟶ `' | "`
escape ⟶ `\ | #`
b.operator ⟶ + | − | × | ÷ | \ | / | ^ | ∧ | ∨
u.operator ⟶ + | − | ¬
relation ⟶ = | ≠ | < | > | ≤ | ≥

Implementation note: Unary and binary arithmetic operators may be lexically distinguished by the absence or presence of termination (a following space).

One more class defines the set of characters that may appear directly within processed text, *i.e.* within a literal character or string:

any.in.quotes ⟶ *any* \ { *quote* | *escape* }

Round brackets above bound a set from which any member may form the production. The class any thus refers to any legitimate UTF-16 printing character except either quotation mark or character used to introduce an "escape sequence" ('\', and '#').

©2006-10, Dr. Ian Robert East

Text within COMMENT and NOTE is more liberal:

text ⟶ *any.in.quotes**

Note that this might produce a null string.

2.1.2 Lexis

Introduction

Unlike more traditional languages, like C or Java, *no* character is discarded. Because program format is strictly controlled by Honeysuckle syntax, *every* character has significance, including each space and 'start-new-line' (SNL). No "white space" is permitted.

Every character is 'consumed' in recognition of some symbol.

Certain symbols, including all keywords, are *terminated*. Termination may take the form of a SPACE, a SNL, or designated punctuation (see below). Any terminating SPACE is consumed.

Again unlike traditional languages, Honeysuckle syntax incorporates annotation, governing where it may occur, and format, including line termination, indentation, and *folding*.

The following symbols lie within the lexis (are recognized by lexical analysis):

– KEYWORD reserved set (excluded from name-space)

– KEY CHARACTER characters used for punctuation, delimitation, and parenthesis

– OPERATOR characters denoting operation or relation

– NAME unique label of entity (process, object, class, *etc.*)
– OPERATOR unique label of unary prefix or binary infix function
– LITERAL value of number, character, or string

– ↲ RETURN commence new line with same indentation
– → INSET … one degree above previous line
– ← OUTSET … one degree less than previous line
– SPACE sometimes explicit within syntax

– COMMENT continues to end of line or end of fold
– FOLD-IN start of a text fold
– FOLD-OUT end of a text fold.

The latter two groups describe format and annotation, respectively.

Space characters may be consumed within RETURN, INSET, or OUTSET symbols, or may be returned as symbols in their own right, where they are not required as delimiters (see below).

NAME, OPERATOR, AND LITERAL, each identify a qualifying attribute – the actual name, operator, or value concerned.

 ©2006-10, Dr. Ian Robert East

Keywords

Keywords are as follows:

a	acquire	after	alias	alternate	always	an
and	any	as	assign	before	borrowed	class
client	collection	create	define	defines	definition	destroy
disable	each	for	from	has	if	imports
in	interface	is	mark	named	network	not
null	of	or	otherwise	parallel	process	provider
quiet	receive	received	repeat	returned	send	sequence
service	signal	skip	stop	terminate	the	then
this	to	transfer	until	value	wait	when
while						

Keywords are *reserved* and cannot thus be confused with *names*. (Lexical analysis may well initially recognize membership of a set enclosing both, later to resolve one from the other.)

Key characters

Various symbols are used for parenthesis, punctuation, and delimitation:

>	?	,	;	:	>	?
(	)	[	]	{	}	\
space						

Operators and relations

Operators may be summarized by the following table:

+	−	×	÷	\	/	^
=	≠	<	>	≤	≥	
∧	∨	¬				

Names

<u>Lexical rule</u>: No name may coincide with any keyword. (Keywords are 'reserved'.)

A name must begin with a letter and end with letter or digit. A name may decompose into a number of words, delimited by a no-break space (NBSP), hyphen, or full-stop (period). There is no restriction upon length.

$$\mathit{name} \longrightarrow \mathit{letter}\ \{\ [\ \text{NBS}\ |\ .\ ]\ \{\ \mathit{letter}\ |\ \mathit{digit}\ \}\ \}^{*}$$

There is no possibility of a hyphen being mistaken for a minus sign because operators and operands are required to be delimited by (breaking) spaces.

Honeysuckle provides only a single universal "name-space", encompassing constants, objects, classes, operations, processes, and services.

Literal values

Honeysuckle recognizes a small set of lexical *categories* ("data types"):

- *number*
  - *natural*
  - *integer*
  - *real*
- *character*
- *string.*

Promotion is permitted via semantic analysis, according to *value classes* defined, but lexical analysis will attribute the least lexical class in the sequence: NATURAL, INTEGER, REAL, which will be recorded as a category of:

$$number \longrightarrow natural \mid integer \mid real$$

By default, the domain of every NUMBER shall be defined by a 64-bit word-width.

To remain applicable to the broadest range of applications, three variants of Honeysuckle are hereby sanctioned:

Honeysuckle	64-bit word-width	HPL	(default)
Honeysuckle-32	32-bit word-width	HPL-32	
Honeysuckle-16	16-bit word-width	HPL-16.	

Recall that byte-ordering is allowed to follow the custom of the platform concerned, and is no concern of language definition.

Implementation note: While mapping to machine representation and verification of machine constraint satisfaction are logically the task of semantic analysis, it serves efficiency to conduct these tasks within lexical analysis.

Semantic rule: Recognition of any NUMBER is subject to the condition that its value lies within a domain of at least one category (NATURAL, INTEGER, or REAL). The lowest consistent category will be recorded.

A *domain violation* error is thus possible, at the lexical level.

The domain of INTEGER is defined by 2's complement representation; that of REAL by the appropriate IEEE standard. (A 32-bit word is employed for each REAL in HPL-16.)

Numbers are decimal by default but provision is made for hexadecimal. Leading zeroes are permitted for either. Lexical analysis reports both token and qualifier; for example, *number* and integer. In this way, it can assist semantic analysis.

$$natural \longrightarrow digit^{+} \mid \{ \ \# \ hex.digit^{+} \ \}$$

$$integer \longrightarrow [+ \mid -] \ natural$$

$$real \longrightarrow [+ \mid -] \ digit^{1+} \ [\ . \ digit^{+} \ ] \ [\ \mathrm{E} \ [+ \mid -] \ digit^{+} \ ]$$

 ©2006-10, Dr. Ian Robert East

hex.digit ⟶ *digit* | `A-F`

Real values cannot be expressed using hexadecimal notation.

Character and string symbols are defined as follows:

character ⟶ `'` *c.character* `'`

string ⟶ `"` *c.character** `"`

c.character ⟶ *any.in.quotes* | { *hex.quad* } |
{ `\` { `n` | `_` | `#` | `\` | `'` | `"` } }

hex.quad ⟶ `#` *hex.digit*⁴

Note that a STRING can be empty but a character cannot.

Recall that ANY is defined by the BMP of UTF-16.

For convenience, we shall define only a scalar LITERAL:

literal ⟶ `signal` | `null` | *number* | *character*

For example, only a scalar can index an array, so we shall consider string (a *vector* literal) as a special case.

SIGNAL represents no value. Its only application is to synchronize processes without passing any value, via:

```
send signal
```

It is thus permitted only within a communication command.

NULL represents the null of some *class*. For example, the null of any numeric class is 0, and that of a list an empty list. In Honeysuckle, every class has its very own null.

Format

The following symbols are recognized through lexical analysis:

↲ → ←

See below for discussion.

Annotation

The following key symbols are used in the process of program annotation:

```
//        {{{//        {{{          }}}
```

Annotation consists of the following symbols, which may be recognized within either syntactic or lexical analysis:

comment ⟶ { *comment.line* | *fold-in* | *comment.fold* } ↲

comment.line ⟶ `//` *space* *text*

fold.in ⟶ `{{{` *space* *text*

comment.fold ⟶ *comment.fold.in* ↲ *fold.out*

comment.fold.in ⟶ `{{{//` *space* *text* { ↲ *text* }*

fold.out ⟶ `}}}`

text ⟶ *any.in.quotes**

The associated semantic rules must also be enforced (see #2.2 below), which will require persistent memory. (Lexical analysis will thus require the equivalent of a *push-down automaton* (PDA), in place of the usual finite-state automaton (FSA).

As with format commands, it is preferable to process (and remove) annotation within lexical analysis, in order to reduce communication with, and simplify, syntactic analysis. While requires a more sophisticated process, it reduces the difficulty in distinguishing TEXT from FOLD-OUT. (The definition of the former must suffer a minor refinement.)

2.1.3 Format

Delimitation, ambiguity, and symbol termination

A single space is the only recognized 'delimiter' (separation between symbols). Single spaces are thus removed by lexical analysis and rendered invisible to syntactic analysis. Multiple spaces are not recognized, except as indentation (see below).

Note that there are no spaces between elements of any symbol in the lexis. Thus, for example, no space is permitted between a prefix '-' and the string of digits making an integer.

There is no such thing in Honeysuckle as "white space", as in C or Pascal, where sequences of spaces and SNLs are simply ignored. Both vertical and horizontal spacing is controlled and thus rendered consistent between programs and programmers.

A SNL (= LF = `\n` = `#000A`) is *not* a delimiter, as it is in C or Pascal. Every SNL should form part of a RETURN, and only thus be consumed.

There is no ambiguity between the terminal symbols of Honeysuckle. Any that might otherwise have arisen – for example, between a name and a keyword – is eliminated by a requirement of *termination*. Most symbols *must* terminate.

The following characters, and mark, are *terminators*:

space SNL

`)` `]` `}` `,` `;` `–`

Only a space is consumed as a terminator. The rest form (part of) a subsequent lexeme.

The following symbols are not themselves terminated:

`(` `[` `{` `:` `–`

↲ → ←

 ©2006-10, Dr. Ian Robert East

The following symbols may or may not be terminated, but only ever by a space:

|

In other words, any trailing space is consumed.

Note: For the sake of simplicity, none of the syntax here details whether a symbol (lexeme) is to be terminated or not, or indeed how. Lexical analysis will require definition of more than one set of terminators.

Indentation and new lines

Each new line must include the indentation necessary to align with the previous one:

$$\hookleftarrow \longrightarrow \textit{return}$$

$$\textit{return} \longrightarrow \textit{full.return} \ \ldots$$

$$\textit{full.return} \longrightarrow \text{SNL} \ \ \textit{indent}^n$$

where n is the current degree of indentation. Lexical analysis must therefore be *adaptive*.

Any indent not absorbed by a RETURN (↲) is termed an INSET.

$$\rightarrow \longrightarrow \textit{inset}$$

$$\textit{inset} \longrightarrow \textit{full.indent} \ \ldots$$

$$\textit{indent} \longrightarrow \textit{space} \ \ \textit{space}$$

Note the implicit definition of an OUTSET (←):

$$\leftarrow \longrightarrow \textit{inset}^{-1}$$

leading to the following implicit productions:

$$\hookleftarrow \ \leftarrow^q \longrightarrow \text{SNL} \ \ \rightarrow^p \qquad n = p + q$$

$$\rightarrow^p \ \leftarrow^q \longrightarrow \varepsilon \qquad (p = q)$$

$$\rightarrow^p \ \leftarrow^q \longrightarrow \rightarrow^{p - q} \qquad (p > q)$$

$$\rightarrow^p \ \leftarrow^q \longrightarrow \leftarrow^{q - p} \qquad (p < q)$$

Lexical analysis must count indents at the start of any line (after any `\n`) and report an appropriate sequence of ↲, →, and ← tokens.

By convention, the parser will only ever encounter the sequence $\leftarrow^+$ ↲. An OUTSET will *always* precede either another OUTSET or a RETURN. An OUTSET will never immediately *follow* a RETURN, nor will an INSET ever immediately *precede* one. (See below.)

In Honeysuckle, only a single INSET will be encountered at a time because indentation is *never allowed to increase by more than one degree on any single line*. (In occam, functions may be defined within an expression, whereupon indentation may become arbitrary.) Because of nesting, multiple OUTSETs on the same line are unavoidable.

Indentation and nesting compound statements

Some care is called for in the composition of syntax with regard to indentation.

First, note that a SNL is just a character, like the two spaces which correspond to an indent in Honeysuckle. INSET (→) and OUTSET (←), however, refer to terminal symbols within the syntax, representing an increase or decrease in the degree of indentation. RETURN (↲) represents starting a new line with the same degree of indentation. We must take care to distinguish the textual (character) and syntactic (symbolic) levels of abstraction.

Implementation imposes the constraint that an OUTSET can only be detected following a SNL. Yet if we force an association between OUTSET and RETURN, we must take care not to generate extraneous blank lines.

Furthermore, we would like to deny any artificial association, as far as possible.

Lastly, whereas we wish, for the sake of transparency, to inset only one degree on any line, we would like to consolidate multiple 'outsets' consecutively, without intervening symbols.

Consider the following syntax, which is free of any artifact, such as an unwarranted RETURN:

$$p \longrightarrow s \mid q$$

$$q \longrightarrow \rightarrow p \; \{ \; \mathtt{a} \; ↲ \; p \; \}^{+} \leftarrow$$

Indentation effectively parenthesizes each *q*, which might represent a sequence, for example. *s* might represent a simple (atomic) process (a single 'action'), and '`a`' a statement separator, like ';' in Pascal. *p* may represent a more general process, simple or compound. (Adding a terminal that introduces the compound statement, on a separate line, changes nothing here, except that it would prevent production of multiple consecutive INSETs.)

Note that the *q* production forces a composition of no less than two 'processes' *p*.

The above productions might spawn:

```
→ s a ↲
  → s a ↲
    s ← ←
```

Here, an OUTSET will always *precede* a RETURN, with a single exception.

The final *s*, of the 'outermost' *q*, will lack a following RETURN. There would thus appear no way by which we might, in practice, detect the two trailing OUTSETs above.

Containers

A solution is to require the 'outer' context to require a RETURN after any *p*. For example, a *container r* might represent the syntax in which a process is named and defined:

$$r \longrightarrow \ldots \; p \; ↲$$

Any such container must end starting a fresh line, though this need not necessarily be blank.

While artificial, it is considered less of an imposition than the only effective alternative, which is to force every atomic statement (*s* above) to terminate with a RETURN. Only the outermost construction is affected.

©2006-10, Dr. Ian Robert East

We now have the ground rules for the definition of syntax using indentation to parenthesize compound statements.

There follows a list of containers identified within Honeysuckle syntax:

– ITEM	
– COLLECTION	
– V-LIST	vertical list
– RECEIPTS	'value' parameter list
– LOANS	'reference' parameter list
– RETURNS	list of objects created
– PAR-INTERFACE	'parallel' interface
– VALUES	list of values (constants) defined
– STATE	list of objects named (variables declared) (applicable to sequential composition)
– NETWORK	prioritized service architecture (PSA). (applicable to parallel composition)

Indentation and hanging characters

Occasionally, syntax invites a character hanging within an INSET or RETURN. An example is a hyphen preceding an item listed within an *alternation* that indicates feedback, or an opening bracket that begins a vertically arranged LIST.

The following variants are verified by lexical analysis:

$$\textit{return'} \longrightarrow \textit{full.return} \mid \{\ \textit{short.return}\ \ \textit{hanging}\ \}$$

$$\textit{inset'} \longrightarrow \textit{full.indent} \mid \{\ \textit{short.indent}\ \ \textit{hanging}\ \}$$

where:

$$\textit{short.return} \longrightarrow \textit{indent}^{n-1}\ \ \textit{space}$$

$$\textit{short.indent} \longrightarrow \textit{space}$$

but only RETURN and INSET are reported for syntactic analysis:

$$\textit{return} \longrightarrow \textit{full.return} \mid \textit{short.return}$$

$$\textit{inset} \longrightarrow \textit{full.indent} \mid \textit{short.indent}$$

Any hanging character would follow:

$$\textit{hanging} \longrightarrow -\ \mid\ ($$

Implementation note: Lexical analysis must be capable of generating a sequence of tokens according to the constitution of a single lexeme.

Implementation note: An intermediate "lexical parser" might recognize both CONTINUATION and RETURN (↵) as these each comprise a sequence of tokens.

Semantic rule: Upon INSET, increment *n*.

Semantic rule: Upon OUTSET, verify $n > f$ (fold inset), then decrement *n*.

Line continuation

Line continuation is indicated via a '\' at the end of the line to be continued and the beginning of the continuing line:

continuation ⟶ *space* \ ↲ \ *space*

A continuation is ignored and does not affect the current degree of indentation.

2.2 Annotation

2.2.1 Comment

Honeysuckle has a single form of comment which occupies an entire line.

comment.line ⟶ // *space* *text*

comment ⟶ { *comment.line* … } ↲

Any comment must match the indentation of any *following* line, and not necessarily any preceding one. This is equivalent to declaring that *no comment can precede an inset*. This will be enforced through further productions.

The terminating RETURN in the above production, eliminates the possibility of any OUTSET below a comment. Therefore, a comment must never appear where an OUTSET is required; for example, as the last line in a compound statement. Syntactic context ensures this.

Honeysuckle syntax also ensures that a terminal always precedes any INSET, eliminating the possibility of any INSET required below a COMMENT.

A secondary concern compelling the inclusion of the COMMENT within Honeysuckle syntax (thus denying the possibility of its removal within lexical analysis) is the intent to confer exclusive ownership of the line on which it sits.

Implementation note: A scanner need report only the presence of a COMMENT, and not its internal structure.

2.2.2 Fold

Honeysuckle offers control over how a program is displayed. Lines may be explicitly *folded*, allowing the program to be navigated hierarchically (as a tree of folds).

FOLD-IN and FOLD-OUT parenthesize lines of text, and must match like brackets:

fold.in ⟶ {{{ *space* *text*

fold.out ⟶ }}}

A FOLD-IN is a direct substitute for a COMMENT, with similar syntax:

comment ⟶ { *comment.line* | *fold-in* | … } ↲

A FOLD-OUT is *not* a comment.

Neither FOLD-IN nor COMMENT may ever precede an OUTSET or INSET. In other words, they are each followed by a subsequent line whose inset they share.

©2006-10, Dr. Ian Robert East

Note that a fold may be empty.

Certain rules apply to the fold and its content:

Semantic rule: A FOLD-OUT must follow an unmatched FOLD-IN.

Semantic rule: Matching FOLD-IN and FOLD-OUT must share a common inset – $n = f$.

No line within a fold may possess an inset less than that of FOLD-IN/OUT. As a result, an INSET must never immediately precede a FOLD-OUT.

Context must always deny the possibility of any INSET preceding a FOLD-IN or FOLD-OUT.

Support for folding thus implies a syntax that affords the opportunity with any entity subject to compound construction to add both a FOLD-IN prefix and a FOLD-OUT suffix.

Semantic rule: A compiler must record the degree of indentation (increment on each INSET, decrement on each OUTSET) and verify each rule above.

Implementation note: Lexical analysis may report each COMMENT, as well as every INSET, OUTSET, and RETURN (→ ← ↲), but need not necessarily distinguish the type of comment (whether COMMENT-FOLD or COMMENT-LINE).

Implementation note Lexical analysis may report each FOLD-OUT, and distinguish each FOLD-IN, only if the parser is to enforce the above rules for folding. However, it is anticipated that this task may be performed within lexical analysis, in which case FOLD-OUT may be ignored (*i.e.* go unreported) and FOLD-IN reported simply as a COMMENT.

2.2.3 Comment-fold

A fold may either reveal or obscure its content. It may constitute either *comment* or *command*. In the former case, it is called a COMMENT-FOLD.

The distinction is marked by the introductory terminal:

comment.fold.in ⟶ `{{{//` *space text* { ↲ *text* }*

Like a command-fold, a COMMENT-FOLD is always terminated by a FOLD-OUT:

comment.fold ⟶ *comment.fold.in* ↲ *fold.out*

Nothing is ever allowed between a COMMENT-FOLD-IN and its closing FOLD-OUT. When a command-fold is converted to a comment-fold, every line within it becomes mere text

A COMMENT-FOLD affords a “block comment” anywhere a COMMENT is permitted.

comment ⟶ { *comment.line* | *fold-in* | *comment.fold* } ↲

All three fold-rules apply equally to both command- and COMMENT-FOLD.

Implementation note: Lexical analysis must afford recognition of any form of COMMENT including any enfolded or terminating RETURN.

HPE requirement: An editor is required capable of toggling the status of any fold. It should be possible to accomplish this in as simple a manner as possible – ideally with a single mouse-click or key press.

The entire content of a COMMENT-FOLD is effectively ignored upon compilation.

Note this allows a block of *arbitrary* size (perhaps thousands of lines) to be removed from compilation ("commented out"), or returned, via a single simple edit (or editor command).

The choice of comment-fold-in mark (`{{{//`) should allow any "free text" folding editor to correctly identify and display folding (without regard to comment-status).

FOLD-IN and COMMENT-FOLD-IN are lexically distinct. However, any HPE should separate fold and comment mark.

HPE requirement: An editor should display the comment-status of a fold, using the line comment mark ('//'), whether open or closed:

– `... // when closed`

– `{{{ // when open`
```
content
}}}
```

Note the space introduced between fold-in and comment marks.

2.2.4 Embedding annotation

Annotation in Honeysuckle is subject to syntactic control in order to:

- avoid 'litter' and render the format of each program similar
- render the inset of annotation and program text consistent.

To be readable, programs need to possess clean and consistent format and annotation.

On the other hand, annotation is invited above any element of a compound statement; for example, any process within a composition or construction.

To achieve this, a COMMENT may often suffix either RETURN or INSET:

$$↲c \longrightarrow ↲ \; [\; \textit{comment} \; \ldots \;]$$

$$\rightarrow c \longrightarrow \rightarrow \; [\; \textit{comment} \;]$$

On many occasions, a FOLD-OUT may also suffix a RETURN:

$$↲c \longrightarrow ↲ \; [\; \textit{comment} \mid \{ \; \textit{fold.out} \; ↲ \; \} \;]$$

ensuring that it both follows an element and shares its indentation.

It may also prefix many an OUTSET:

$$f\leftarrow \; \longrightarrow \; [\; ↲ \; \textit{fold.out} \;] \; \leftarrow$$

allowing the final line in a construction to be included within a fold. (A COMMENT-FOLD will in any case encompass a FOLD-OUT.)

An example of construction that incorporates folding follows:

$$\textit{parallel} \longrightarrow \texttt{parallel} \; ↲ \; \rightarrow c \; \textit{process} \; \{ \; ↲c \; \textit{process} \; \}^{+} \; f\leftarrow$$

 ©2006-10, Dr. Ian Robert East

Implementation note: If lexical analysis absorbs responsibility for the verification of the rules of folding, it need report neither COMMENT nor FOLD-OUT.

Note that a single minor edit, changing a FOLD-IN to a COMMENT-FOLD-IN, changes program syntax (parsing) substantially, absorbing all enfolded text into a single production.

Requirements for Honeysuckle Programming Environment

Character set

A text editor is required that can accommodate an adequate character set. This goes beyond the set defined by the American Standard Code for Information Interchange (ASCII).

The set defined by the 16-bit Unicode Translation Format (UTF-16) is adequate.

Folding

A folding editor is required to properly portray the structure of a Honeysuckle program. It should enfold any region surrounded by *fold-marks*.

The Honeysuckle document model should run as follows: A *document*, or a *fold*, is a sequence of *lines*. A line is either a *line of text* or a fold. Any line (and thus fold) has a characteristic *inset* (degree of indentation).

Should the inset of any fold be changed, that of every line within it is similarly affected.

A fold may be displayed *open* or *closed*. Entering a fold causes a (or the) window to be display its contents exclusively. The following operations thus apply to a fold: *add*, *remove*, *enter*, *exit*, *open*, *close*.

A folded section is displayed, for example:

```
sequence
  ...
```

whereas an explicitly folded region appears with the remainder of the fold-line:

```
...  here lies a closed command-fold

{{{ and here is an open (though empty) command-fold
}}}
```

The level to which sections are folded should be easily controlled.

Any fold must easily be convertible to a *comment-fold* (a Honeysuckle block-comment), preferably with a single mouse-click or key-press.

The status of any fold (comment- or command-) should be indicated via a leading mark on the fold-line. For example:

```
... // a closed comment-fold

{{{ // an open (though empty) comment-fold
}}}
```

Honeysuckle syntax allows the compiler to be aware of fold status. It must react accordingly and ignore any command in a comment-fold. Only the fold-in mark need be altered to effect a change in fold status. There is no need for an editor to modify fold content in any way.

©2006-10, Dr. Ian Robert East

3 Project composition

3.1 Item

Honeysuckle interposes “clear blue water'’ between system and project modularity. Each definition of process, object, and service, is termed an *item*. Items may be gathered into a *collection*. Items and collections serve the needs of separated development and reuse.

A program consists of one or more items, including at least one process. For example:

```
definition of process greet

  note
    ©2005 Ian East

  imports
    from Environment
      service console
      class String

  process greet is
    {
      note
        display a fixed message

      interface
        client of console
      define
        String greeting is "Hello world!\n"

      send greeting to console
    }
```

A process may instead be defined *in-line* with identical semantics. Any further (offline) definitions must then be imported above the description of the parent process. An in-line definition is achieved simultaneously with command issue (*greet!*).

```
...
{
  ... context

  send greeting to console
}
...
```

A process thus defined can still be named, facilitating recursion. (See Section 5.1.) Recursion is also permitted in an off-line definition.

Sharing definitions between applications requires a collection.

One consequence of the separation between project and system modularity is that there is the minimum connection between language and data type definition. The latter is termed *class* definition. The Honeysuckle *system environment* (HSE) must provide a set, standardized by convention only.

The following productions suffice:

```
               item  ──→  definition of item.declaration ↲ ↲
                          →c comment [ importation ] definition f← ↲
‡         item.type  ──→  value | class | process | service
        importation  ──→  ↲ imports ↲ →c { import { ↲c import }* f← ↲
‡            import  ──→  { item.declaration [ from name ] } |
                          { from name ↲
                          →c { item.declaration { ↲c item.declaration }+ f← } |
                          { collection name }
   item.declaration  ──→  item.type name
         definition  ──→  ↲ { value.defn | class.defn | process.defn | service.defn }
‡        value.defn  ──→  name value name … is evaluation
‡        class.defn  ──→  class name [ of name ] class.construction
       process.defn  ──→  process name is ↲ → process ←
       service.defn  ──→  service name is ↲ → service ←
```

A blank line divides the three elements in an item definition.

Note the distinction between the definition of value and class from that of process and service. Elements of the former may be contained within a single line, whereas those of the latter begin on a new line. The distinction reflects their relative importance.

Inheritance of type applies to value only; hence, the omission of a leading name in the productions governing the definition of class, process, and service.

See Section 4.3 below for a rule applicable to the import of an entire collection.

All item definitions in Honeysuckle are *transparent*. For example:

```
definition of service Console
  imports class String from StandardTypes
```

Any definition that imports service Console, automatically imports class String, and thus need not import it directly.

Semantic Rule: The type of item defined must match that announced.

Semantic Rule: Importing an item or collection already imported is disallowed.

This is to avoid unintended collisions between different items, similarly named.

 ©2006-10, Dr. Ian Robert East

3.2 Collection

Collections allow the gathering together of definitions that are related according to their application or function. They thus afford *project modularity* and provide for separated development and reuse.

```
collection driver

  {{{// notes
    version
    ©date, author
    version history
    implementation notes
  }}}

  imports
    ...

  defines
    process read
    process write
    ...

  process read is
    {
      ...
```

In no sense is a collection a means of ‘object-based’ programming, like a module in Modula-2, a unit in Delphi Pascal, or a file in C. The only scope defined within a collection is that of item definitions, either imported or rendered locally. A collection has no state nor does it possess any influence over the design of a system. It simply serves as the means by which a project may be divided into packages for separated development, test, and reuse.

Item imports and definitions are identical to those within individual items.

‡ *collection* ⟶ `collection` *name* ↲ ↲
→c *comment* [*importation*] [*manifest*]
definition { ↲ *definition* }*
f← ↲

‡ *manifest* ⟶ ↲ `defines` ↲
→c { *item.declaration* { ↲c *item.declaration* }+ *f*← ↲

<u>Semantic Rule</u>: The list of definitions in a collection must match that listed in its manifest.

Note that a collection need not define anything. It may simply act as a node in a *collection hierarchy* – corresponding to the familiar top-down decomposition or “structure chart” – recording the way in which definitions are hierarchically collected within a particular project.

Mutual recursion must be between defined item types that share scope within a collection.

3.3 Project

Composition

Items and collections may be composed via importation to form a *project*.

Interdependency between items and collections, compilation and version history, and virtual and machine code, relating to a project is recorded within a single *project file*. How this is done remains specific to the implementation of each *programming environment*.

Importation is *transparent* in the sense that any imported item can carry with it all definitions it imports itself, and so on. This would imply a "paper-clip" effect, where picking up a single item may bring the entire box. It would invite "name-collision". However, a previously imported item is only *available*. If not explicitly imported, its name can safely be reused.

Caution: A different rule applies to the importation of an entire collection. Reference is then required to *every* member of an imported collection. Failure so to do is an error.

A Honeysuckle collection is divided into (source code) interface and implementation sections. Only the former is visible to a potential importer. Only item *definitions* may be exported. Neither object nor process can be declared or created within in a collection. This occurs only within a process definition. Only a process can *own* objects or other processes. A program module (item or collection) owns only their definition.

Implementation note: Any project must contain at least one ITEM or COLLECTION.

The minimum project must comprise at least one of either kind. If this is a collection, at least one element must be a process for it to be executable.

An ITEM or COLLECTION encapsulates any compilation.

Implementation note: Syntactic analysis may always be completed without encountering end-of-file (EOF). Any such encounter (within lexical analysis) should thus be reported as an error. (A parser should cease requesting tokens from the scanner when an item or collection is complete.)

Honeysuckle syntax incorporates annotation, which includes *folding*.

Fold-marks occur in pairs – a FOLD-OUT must follow a FOLD-IN – and share a common inset (degree of indentation).

As a result, syntactic analysis must incorporate the creation and maintenance of a *fold-stack*, which must be empty upon completion of an ITEM or COLLECTION.

Implementation note: No ITEM or COLLECTION is complete until the fold-stack is empty.

Recommendations for Honeysuckle Programming Environment (HPE)

- render visible all definitions available at any point of importation
- render visible the sequential and parallel interface of any individual process defined
- support *folding* to render structure apparent and encapsulate (*e.g.* import) lists.

 ©2006-10, Dr. Ian Robert East

Requirements for Honeysuckle System Environment (HSE)

- provide classes *Boolean*, *natural*, *integer*, *real*, *character*, *time*, and *string*.

4 System design

4.1 Introduction

4.1.1 System modularity

The design of a concurrent/reactive system may be expressed in Honeysuckle, prior to, and independent of, any implementation.

Each design is described purely in terms of communication protocol, elaborating the *dependency* of services provided upon services consumed.

With Honeysuckle, the term 'system' is reserved for something that is complete in itself, and therefore has no interface. Everything else is regarded as a *component*.

A component will possess an *interface* comprising services provided or consumed externally. These replace the conventional (data-flow oriented) notion of 'input' and 'output'.

A complicated system, composed of many interdependent services may be described and visualized by hierarchical decomposition. Each component may be initially described according to its interface. A design may then be progressively refined until that of each component is *complete*.

A design (of system or component) is complete when that of all its components is complete. Only then may it be verified and implemented.

HPE requirement: An HVDL editor should offer the ability both to abstract a component, by depicting only its interface, and to *expand* it by progressively revealing more of its internal composition (prioritized service architecture).

4.1.2 System composition

Kinds of composition

A *process* implements both system and component. Design requires a *composition*:

process ⟶ … *composition* …

Each composition may combine components in either sequence or parallel.

composition ⟶ { ↲
→c *interface values*
{ { *state* ↲ *sequence* } | { *network* ↲ *parallel* } }
f← ↲
}

©2006-10, Dr. Ian Robert East

sequence ⟶ `sequence` ↲ →c *process* { ↲c *process* }$^+$ f←

parallel ⟶ `parallel` ↲ →c *process* { ↲c *process* }$^+$ f←

Braces bracket each composition with a definition of its *context*.

Context of composition

The *context* of a composition is the set of all entities to which reference is made.

It may declare objects (variables), define values (constants), and render explicit a formal interface, comprising a sequential part and/or a parallel part:

interface ⟶ [*seq.interface*] [*par.interface*]

A *sequential* interface comprises values received, and objects borrowed, from its parent. Objects may also be returned to the parent, following their creation.

seq.interface ⟶ [*receipts*] [*loans*] [*returns*] ↲ ↲

A *parallel* interface comprises services provided or consumed by other components in its environment.

par.interface ⟶ `interface` ↲ →c … f← ↲ ↲

Note that any composition may have *both* a sequential *and* a parallel interface.

On the other hand, composition is *either* sequential *or* parallel. Process context will possess either common object (naming) or network declaration; never both.

The context of a sequential composition may include the naming of *objects* to which access is shared. That of a parallel composition may include the declaration of *services* rendered between components, forming a *network*.

network ⟶ `network` ↲ →c … f← ↲

A parallel composition *must* define a network. A sequential one *must* declare objects.

A composition is, by definition, a set of components which communicate with each other.

Semantic rule: Reference to any object created by a component of a parallel composition must be confined within that component. It must *not* occur in any other.

Composition syntax is described in greater detail in Section #5.2.2 below.

In Honeysuckle, design is almost entirely concerned with *parallel* decomposition. The term 'component' will henceforward refer only to a member of a parallel composition.

It is considered easier to understand the syntax of the NETWORK statement first, and then that of the INTERFACE statement later, so this is the order in which they are documented below.

4.1.3 Visual and textual design

A design may be expressed in either textual or graphical form. The subset of Honeysuckle syntax employed is called the *Honeysuckle Design Language* (HDL). The collected graphical

constructions form the *Honeysuckle Visual Design Language* (HVDL). Both are documented below, and summarized in Appendix B.

As with the textual language, HVDL is defined down to the finest detail, including the orientation and the specific colour to be employed for every element. See Appendix B.

This document is currently published in two colours only (black and white). Even when a colour version is made available, correct reproduction cannot always be guaranteed. As a result, colours will be defined using 24-bit RGB enumeration.

A one-to-one correspondence exists between textual and visual expression.

Visual expression is not be obligatory, but may be offered as a means of *generating* the textual form. Every visual design must satisfy all design rules.

HPE requirement An HVDL editor must be capable of translation (in either direction) between textual and visual representations, and must not admit any network in breach of a design rule.

Design rules earn their name by being verifiable without any specified implementation. Rules applicable to an implementation are termed *conditions*.

4.1.4 Independent service

Textually, a network is defined, under the NETWORK heading, as part of the context of a parallel composition within a process definition.

Pictorially, a service is represented by an arrow.

```
network
  s
```

Figure 1 A single independent service.

The simplest possible network comprises a single service. Service provision is *independent*, by which we mean no other service need be consumed to enable its provision. Visual and textual representation is shown above

‡ *network* ⟶ `network` ↲ →c … *n.element* { ↲c *n.element* }$^+$ *f*← ↲

‡ *n.element* ⟶ { *provision* | … }

‡ *provision* ⟶ … *s.name*

‡ *s.name* ⟶ *name* …

An internally consumed service can be distinguished from one forming part of a component interface graphically by a thin line terminating an arrow (see Fig. 1).

External consumption or provision is indicated by the same thin line drawn *across* an arrow, near one end or the other accordingly. No arrow may be crossed or terminated by more than one line. (Every system or component comprises at least one process.)

©2006-10, Dr. Ian Robert East

Figure 2 External consumption, internal provision

Figure 3 External provision, internal consumption.

Textually, there is no need to make any such distinction because a separate INTERFACE definition delineates services externally consumed or provided.

4.2 Service dependency

4.2.1 Dependency

Within any component or system, the provision of one service may be *dependent* on the consumption of another. Dependency implies defining an *interface* between services.

Each dependency may be *explicit* (direct) or *implicit* (indirect), with an internal network defined. A network is *final* when all its implicit dependencies are reduced to explicit ones.

Any network (design) that possesses an *implementation* will be considered final.

<u>HPE requirement</u>: A Honeysuckle Programming Language (HPL) compiler is only required to parse a single NETWORK statement. A HDL or HVDL interpreter, on the other hand, *is* expected to parse a hierarchy of network definitions.

The designer may thus begin with an incomplete design, where some or all components are described by their interface only, then progress to a complete design, which can be verified, and then repeatedly refine the verifiable design by further elaborating each interface.

The nature of a dependency falls into one of two distinct categories.

Figure 4 A single interstitial dependency.

Figure 5 A single interleaved dependency.

An *interstitial* dependency is completely consumed between two successive communications of the dependent service. Consumption is completed before provision may continue.

In contrast, an *interleaved* dependency sees its communications interleave with those of the dependent service. There may be more than one. Provision and consumption of all services concerned proceed concurrently. All processes involved are thus concurrent.

This distinction becomes important with regard to *feedback* and *interleaving* (of provision).

Note that the nature of each dependency – interleaved or interstitial – is an attribute of the *interface*, and not of any service *per se*.

Textually, each dependency is expressed on a single line using a binary operator ‘>’:

```
network
  s1 > s2
```

```
network
  s1 > |s2
```

Interleaving with the dependent service is indicated by a qualifier (a preceding '|'). Interstitial consumption is otherwise presumed.

‡ *n.element* ⟶ { *provision* | *dependency* | … }

dependency ⟶ *provision* > *consumption* …

consumption ⟶ [|] *name* …

Dependency upon multiple services merely requires them listed.

```
network
  s1 > s2, s3
  s2 > s4
  s3 > s5
```

Figure 6 Multiple, chained, dependency.

Semantic rule: Any dependency may be expressed only once.

The above graphical and textual syntax denotes a dependency upon multiple services which, while they do not interleave with the dependent service, *may* interleave with each other.

It is sometimes desirable to indicate that consumption does *not* interleave:

```
network
  s1 > s2; s3
```

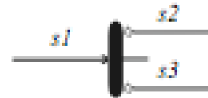

Figure 7 Sequential consumption.

Consumption is then explicitly sequential.

The ';' indicates that the preceding service is completed before the following one starts.

dependency ⟶ *provision* > *consumption* { { , | ; } *consumption* }*

The distinction is valuable with regard to feedback within interleaved provision (see below).

Sequential interstitial consumption implies that each service is completely consumed within distinct interstices within the service provided.

Implementation note: Parsing must include detection of services consumed within the same interstice within the service provided.

Similarly, we can indicate graphically explicitly that consumption *does* interleave:

```
network
  s1 > s2, s3
```

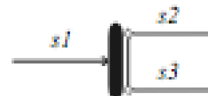

Figure 8 Interleaved consumption.

In other words, services are consumed in the same interstice within the service provided.

There is no such clarification when multiple dependency is expressed textually. Interleaved consumption should therefore be assumed when services are comma-separated.

©2006-10, Dr. Ian Robert East

4.2.2 Repeated dependency

A *chain* may be constructed by horizontal replication of a named service:

```
network
  repeat for 2
    s1 > s1
  s1:2 > s2
  ...
```

```
network
  repeat for 2
    s1 > s2, s3
    s2 > s1
  s3 > s4
  ...
```

Multiple instances created in a horizontal replication may also suffer *design asymmetry* (see below). Each may be individually distinguished via an index (beginning with unity (1)).

‡ *n.element* ⟶ { *provision* | *dependency* | *chain* … }

‡ *chain* ⟶ `repeat for` *value* ↲
→c *dependency* { ↲c *dependency* }* f←

‡ *s.name* ⟶ *name* [: *natural* …]

Note that only *provision* need be distinguished by an index.

By default, the final instance is deemed to possess *no* dependency.

Recall that no circuit is allowed in any chain of dependency.

4.2.3 Asymmetric dependency

A *design asymmetry* occurs when dependency in the provision of two instances of the same service differs. An example might be formed were S3 in Fig. 6 replaced by a instance of S2. This would make it impossible to document dependency without ambiguity. (Note that no such ambiguity would result upon implementation since each component interface could be matched according to dependency. A compiler could still compose components correctly.)

An *implementation asymmetry* is where two instances of the same service are not interchangeable, even though there may be no design asymmetry. Some relationship between information exchanged is material to the system required.

Since any service is either shared or distributed symmetrically, any asymmetry (in design or implementation) prohibits both sharing and distribution.

An asymmetry may be introduced by declaring a *service alias*. Thus the asymmetric variation on Fig. 6 could be expressed by beginning the network declaration with:

```
network
  named
    s2 : s3
  ...
```

‡ *network* ⟶ `network` ↲
→c … [*state*] *n.element* { ↲c *n.element* }$^+$ f← ↲

Components may be composed with no knowledge of any service alias. Furthermore, to be reusable between applications, they must be defined with reference only to the underlying service type, and not to any specific alias.

To resolve this issue, the interface declaration of any component, whether in-line or off-line (as a separate item) may accept an incoming parameter via an ALIAS clause. (See earlier discussion under *parallel interface*.)

4.3 Aggregated service provision

4.3.1 Mutually exclusive provision

Service provision may be composed in two distinct ways. First, services may be grouped into a *bunch* according to *mutual exclusion*. This is expressed as follows:

```
network
  select
    s1
    s2 > s4
    s3
```

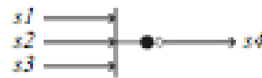

Figure 9 A mutually exclusive *bunch* of services.

Any dependency of any member is also a dependency of the entire bunch. This is because every member may be forced to await consumption of the dependent service.

The textual form can easily indicate when that dependency extends to more than one service of the group. A pictorial expression is facilitated by a short extension of the corresponding line. Alignment with the interface connection also indicates dependency. For example:

```
network
  select
    s1 > s4
    s2
    s3 > s4
```

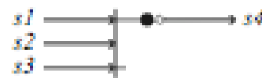

Figure 10 Multiple dependency within mutual exclusion.

Bunching under mutual exclusion is both associative and commutative. Honeysuckle denies both chaining and (visual or textual) nesting of a SELECT construction.

‡ *n.element* ⟶ { *provision* | *dependency* | *chain* | *selection* | … }

selection ⟶ `select` ↲ →c *n.s.element* { ↲c *n.s.element* }⁺ *f*←

n.s.element ⟶ { *provision* | *dependency* }

Providing the same service more than once under mutual exclusion is never necessary, given the option of declaring any service *shared* (see below), and is thus denied.

Because exclusive construction is commutative, the service digraph may be re-arranged by changing the order in which mutually exclusive services appear. This may prevent dependency chains from crossing over, and thus dissolve an apparent threat of deadlock.

Bunching (mutual exclusion) is implemented via the *selection* construct.

©2006-10, Dr. Ian Robert East

4.3.2 Alternation – interleaved provision

The second way of composing service provision is to allow it to *alternate* (Fig. 10).

```
network
  alternate
    s1
    s2 > s4
    s3
```

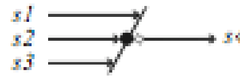

Figure 11 Alternation of service provision.

An alternation represents a second occasion when services *interleave*. It may also be referred to as *interleaved provision*.

Alternating services are *prioritized* according to their vertical arrangement. Any service ready to proceed will *pre-empt* any listed beneath it and currently in progress. It will continue until either *completion* or until interrupted by one of yet higher priority.

‡ *n.element* ⟶ { *provision* | *dependency* | *chain* | *selection* | *alternation* }

‡ *alternation* ⟶ `alternate` … ↲
→c *n.a.element* { ↲c *n.a.element* }$^+$ *f*←

‡ *n.a.element* ⟶ … { *provision* | *dependency* | *selection* }

Selection (bunching) and interleaving of provision is described (and depicted) vertically, in contrast with dependency, which is expressed horizontally.

Multiple dependencies, within any interleaved interface, are drawn extending vertically:

```
network
  alternate
    s1
    s2 > s4, |s5
    s3 > s6
```

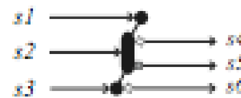

Figure 12 Interleaved provision with multiple dependency.

Nesting of one ALTERNATE construction inside another is unnecessary and denied.

SELECT within ALTERNATE is permitted, but not ALTERNATE within SELECT.

Like bunching, an alternation exhibits purely sequential behaviour. It is implemented using the WHEN construction (#5.2.3).

Unlike bunching, any dependency affects only the service to which it connects, and no other in the composition. To protect such independence (and avoid the potential for deadlock):

Semantic rule: No two alternating services may depend upon a shared service.

In summary, communication may be interleaved in three different (orthogonal) ways: according to provision (to the left of an interface), consumption (to the right of an interface), and dependency (across an interface).

Repeated interleaving can be indicated using *vertical* replication:

```
network
  alternate for 2
    select
```

```
s1
s2 > s4
s3
```

Subsequent reference to replicated services is via indexing, as with those in a chain.

‡ *alternation* ⟶ `alternate [ for` *value* `]` ↲
→c *n.a.element* { ↲c *n.a.element* }$^+$ *f*←

This construction affords an interface where a common service is provided to multiple clients with a varying degree of priority.

Note that at most two indices are needed to refer to the provision of any instance of service.

‡ *s.name* ⟶ *name* [: *natural* [: *natural*]]

4.3.3 Feedback

Feedback of dependency is possible across an interleaving, subject to three constraints, expressed within Design Rule 4. (See #1.4.4.)

These ensure the threat of deadlock remains excluded.

Feedback is indicated visually by a single arrow running backward (right-to-left).

```
network
  alternate
   -s1
    s2 > s1-
```

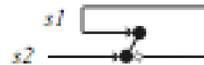

Figure 13 A feedback dependency chain.

A hanging indent, containing a hyphen, gives explicit textual indication of local consumption (within the construction). A corresponding trailing hyphen indicates local provision.

‡ *n.a.element* ⟶ [-] { *provision* | *dependency* | *selection* }

‡ *consumption* ⟶ [|] *name* [-]

(See #3.2.2 for how a hanging indent is recognized and reported.)

Neither provision nor consumption of a fed-back service can appear within the corresponding INTERFACE declaration.

Care is needed with regard to the semantics of an interleaving of service provision. An operational semantics is required for implementation. As with any translation, resolution between models for abstraction can sometimes require special consideration.

When a feedback chain is of length one, and the dependency concerned is interstitial, producer and consumer execute concurrently. Each corresponding pair of send/receive commands may be executed as assignments.

Implementation note: When the feedback chain consists of a single dependency (*i.e.* is of length one), a special method of code generation will be required, replacing each SEND/RECEIVE combination with assignment.

This lies in conflict with an operational model where "a higher-priority process pre-empts one of lower priority". Service, not process, is subject to prioritization.

 ©2006-10, Dr. Ian Robert East

Design rules are applicable to interleaved provision, and to feedback within it. (See #1.4.4.)

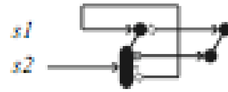

Figure 14 An example of legitimate consumption sequential to a feedback path.

4.3.4 Parameter passing and dynamic configuration

Any replication (horizontal or vertical) may be bounded by a *configuration value* passed as a parameter. Since this may be computed upon reference (invocation), the composition of a network may be subject to *dynamic configuration*.

‡ *network* ⟶ `network` ↲
→c [*receipts*] [*state*] *n.element* { ↲c *n.element* }$^+$ *f*← ↲

See #5.2.5 for how to pass configuration values.

Their reception is similar to that of *processed values*, discussed previously:

```
network
  received Natural Length
  alternate for Length
    s
```

4.4 Shared, synchronized, and distributed, service

4.4.1 Sharing, distribution, and system design

Any service may be shared among clients, distributed across providers, or both shared and distributed. This affords many-to-one, one-to-many, and many-to-many, interconnection.

Sharing and distribution are attributes of the *use* of a service, and not of either the service itself, or of the interface to which it is connected. Declaration of either, or both, thus pertains only to the definition of a particular network.

Whether or not a service is shared or distributed is a concern of *design* only, and not one of implementation. A component's interface is described without reference to either issue. Consideration is necessary only when combining components.

Normally, a service represents a contract between two parties only. Sharing or distribution extends the contract to involve more than two. A similar effect is introduced by the shared channels of occam 3 [9].

A third attribute of the use of a service occurs where sharing is *synchronized*. Here, a number of distinct clients are each served once before the cycle can begin again. A notion of "fairness" may thus be introduced, ensuring no member of some set of consumers goes 'hun-

©2006-10, Dr. Ian Robert East

gry' while others monopolize service. An exclusive resource may be protected, or a 'barrier' created, punctuating repeated execution of a given set concurrent processes.

Implementation note: Clients of a synchronized service must be distinguishable.

While it is conceivable that a service be both distributed *and* synchronized, it is nonetheless undesirable. It would introduce unwarranted complexity in both syntax and abstraction.

In Honeysuckle, a service is either synchronized *or* distributed – provision is then either purely sequential or purely concurrent. Only *un*synchronized sharing may describe a mix of both sequential and concurrent behaviour.

4.4.2 Sharing

Any service may be shared among multiple clients. Whether or not provision is indeed shared is rendered implicit by the pattern of consumption.

Textually, where there is no dependency, sharing is indicated only in implementation. Pictorially, *convergence* is employed, towards a service arrowhead.

Figure 15 A single service shared by two internal clients.

Sharing replaces multiple exclusive provision of the same service. If the same service is to be consumed by more than one process, a single instance *must* be declared shared *unless* consumption is prioritized.

In sharing access, a client may face a delay, while another is served. Once service begins, however, it will be exclusive.

Implementation note: Sharing requires a queue, rather like entry to a monitor, together with an additional rendezvous location to synchronize access.

Honeysuckle guarantees 'fair' service, in the sense that any request for any shared service *will* eventually be granted. The protocol of a "first-come-first-served" queue applies.

4.4.3 Synchronized sharing

Honeysuckle affords *synchronized sharing*, where every client must consume a service once before any can reinitiate consumption, and the cycle begin again.

Figure 16 Bulk synchrony with synchronized sharing.

Like sharing, synchronized sharing is 'superstructure'. It could be implemented directly via the use of an additional coordinating process but is believed useful and intuitive enough to warrant its own syntax. The level of system abstraction possible is raised.

 ©2006-10, Dr. Ian Robert East

Implementation note: Synchronized sharing requires a secondary queue from which elements are prevented from joining the primary one until a cycle is complete.

Like sharing, synchronized sharing is not an attribute of either service or process. It refers only to how a service is provided within a particular network design.

When a synchronized service forms part of a component interface, the number of clients served must be rendered explicit. This is known as the *degree*.

‡ *provision* ⟶ [{ `synchronized` | … } [*degree*]] *s.name*

‡ *degree* ⟶ [*natural*]

Semantic Rule: The degree quoted must exceed one.

Where a synchronized service is consumed entirely within a component, and is thus not part of its interface, attributing a degree to provision is optional.

4.4.4 Distribution

Sharing affords many-to-one interconnection. Distributed provision allows many-to-many.

A single service may connect multiple providers and multiple consumers:

```
network
  distributed validate
```

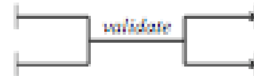

Figure 17 A single service both shared and distributed.

When a service forms part of a component interface, the degree of distributed provision must be stated. Textually, a natural number is placed between distributed and the service name. Graphically, the service arrow is crossed and a natural number placed above.

Within a network, neither the textual nor the graphical description of distribution need render explicit the degree. In implementation, there may be any number of providers. However, where distinct providers are rendered graphically explicit, their number *is* defined by design.

‡ *provision* ⟶ [{ `synchronized` | `distributed` } [*degree*]] *s.name*

Within a network, it can never make sense to offer more providers than consumers.

Semantic rule: Where a service is consumed internally (entirely within a component), the number of providers must not exceed the number of consumers.

A synchronized shared service may not be subject to distributed provision.

From the external perspective, component behaviour is not significantly affected when provision of a dependency is shared and/or distributed. Consumption is, at worst, delayed.

The architecture below might belong to a component that secures the release of a resource by requiring multiple clients to perform an identical protocol.

```
network
  synchronized [3] claim
  request > claim
```

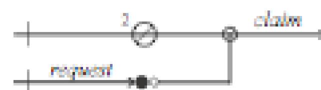

Figure 18 Synchronized sharing for resource protection.

©2006-10, Dr. Ian Robert East

Any synchronized service requires *all* its consumers to be served once before any are served again. There is no facility for exemption. (It remains a possible future language refinement.)

Multiple providers of a dependency may also be expressed:

```
network
  approve > validate
  distributed validate
```

Figure 19 Dependency upon a service with distributed provision.

Note that a service cannot be both distributed and fed-back. Distributed providers must *all* be independent of the consumer.

4.4.5 Avoiding "crossed service" and resource allocation deadlock

Both exclusive and shared service provision open the possibility of *resource allocation deadlock*. This can occur when service consumption is interleaved.

Design Rule 2 exists to deny this possibility. (See #1.4.4.)

Recall that it is not permitted to provide two mutually exclusive instances of the same service. Instead, a single provision must be shared. Consumption of a shared service proceeds exclusively (without interleaving). As a result, deadlock cannot arise.

4.5 Modularity and the component interface

4.5.1 Introduction

Honeysuckle affords *modular* design. A design can be reduced to a set of distinct *components*, specified by their INTERFACE. The interface of each component describes *what* it does, and is expressed entirely by the services provided and consumed.

A design may be progressively refined by qualifying *how* services provided depend upon those consumed. This is achieved by describing the internal NETWORK, which may include internal components that are merely specified, and not yet designed. In other words, they possess an INTERFACE, but not yet a NETWORK, statement.

HVDL and HDL continue to indicate modularity at every stage of refinement.

HPE requirement: Any HVDL editor should be able to display any design at *every* level of refinement, *i.e.* with components either designed or merely specified. The 'top-most' level might contain a single 'empty' component. The subsequent level would describe a design (network), which might include further 'empty' components, and so on until the design is *complete*..

Within an interface, service provision may be qualified according to whether it is:

- *concurrent* internally distributed among multiple providers
- *synchronized* each consumer is served once before any is served again

 ©2006-10, Dr. Ian Robert East

(synchronization may extend across multiple services)

– *prioritized* service will pre-empt that to others of lower priority (replication of a prioritization is possible).

Each of the above must also be quantified. The *degree* of concurrency, synchronization, or priority must be added. (The degree of synchronization infers the number of consumers that must be served in each cycle.)

Any provision may be *shared* among multiple consumers.

Multiple services provided may or may not be exclusive. If exclusive, selection is always deterministic, though it may be decided according to either state internal to the component or to state residing within the component interface (within a service in that interface).

A design where every component is both specified and designed is termed *complete*.

Only a complete design can be verified free of any potential deadlock.

4.5.2 Service provision and consumption

An interface documents the *ports* via which a component can be connected to a network. Each port forms one end of a service – either client (consumer) or server (provider).

Ports are declared under an INTERFACE heading, which forms part of the context of a process.

‡ *par.interface* ⟶ `interface` ↲ →c *p.i.element* { ↲c *p.i.element* }* *f*← ↲

‡ *p.i.element* ⟶ { *port* … }

‡ *port* ⟶ { `provider of` … | `client of` } *service.id*

‡ *service.id* ⟶ *name* …

An interface may be expressed graphically or textually. For example:

Figure 20 Visual representation of component interface.

No port is named. It is sufficient to refer to the *service* concerned.

Nothing is said about the relationship between 'input' and 'output'. This aspect of function is left to the NETWORK statement. A complete *specification* requires a complete interface and an *abstract* definition of network.

Nothing need be said about the relationship between two distinct services provided (at 'input'). Provision may or may not be exclusive.

Ports may be inherited from the parent composition. If the parent composition is sequential, they must form part of its declared parallel interface; if parallel, then they may either form part of its interface or connect services within its network.

©2006-10, Dr. Ian Robert East

When an interface is defined in-line, a known alias may be assigned.

service.id ⟶ *name* [`alias` { *name* | ? }]

When defined off-line (as a separate *item*), an *unknown alias* may be indicated by a question mark. The alias is assigned via the parameter mechanism. For example:

```
parallel
  ...
  mediate ; s2
  ...
```

```
process mediate is
  {
    ...
    interface
      client of s1 alias ?
```

4.5.3 Concurrent provision

A degree of *concurrency* is possible in the consumption of a service when it is *distributed* among a number of internal providers. For example:

```
interface
  provider of [4] s1
  client of s2
```

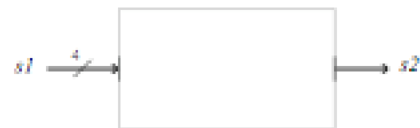

Figure 21 Distributed provision within an interface.

Here, up to four consumers of *s1* may be connected before a queue must form and a measure of sequential consumption arise.

‡ *port* ⟶ { { `provider of` [*degree*] } |
{ `client of` } } *service.id*

Semantic rule: Any degree is a natural number that must exceed one.

4.5.4 Synchronized provision

Service provision may be *synchronized*, in that a number of consumers must each receive service before any may be served again.

```
interface
    synchronized [2] provider of s1
    client of s2
```

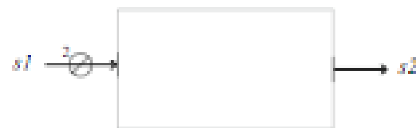

Figure 22 Synchronized provision within an interface.

Here, two distinct clients must each consume *s1* before either may re-initiate service.

‡ *p.i.element* ⟶ { *port* | *synchronization* … }

‡ *synchronization* ⟶ `synchronized` { *degree* *provider* } | ↲

 ©2006-10, Dr. Ian Robert East

→c *n.provider* { ↲c *n.provider* }$^+$ *f*←

‡ *n.provider* ⟶ [*degree*] *provider*

‡ *provider* ⟶ `provider of` *service.id*

Honeysuckle also allows synchronization of *distinct* services.

```
interface
    synchronized
      [2] provider of s1
      provider of s2
    client of s3
```

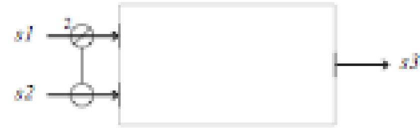

Figure 23 Synchronization of distinct services..

Here, *s1* must be consumed twice, and *s2* once, before either may be consumed again.

While elements of a synchronized provision might, in theory, be consumed concurrently, they are, in Honeysuckle, consumed *sequentially*, in order to simplify the model for abstraction.

4.5.5 Prioritized alternating (interleaved) provision

Service provision may alternate between a number of clients, according to a prioritization.

‡ *p.i.element* ⟶ { *port* | *synchronization* | *prioritization* }

Honeysuckle also allows prioritization of distinct services.

```
interface
    prioritized
      [2] provider of s1
      provider of s2
    client of s3
```

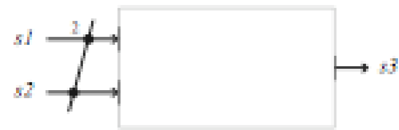

Figure 24 Prioritized alternation of provision.

Syntax is thus similar to that expressing synchronized provision:

‡ *prioritization* ⟶ `prioritized` { *degree provider* } | ↲
→c *n.provider* { ↲c *n.provider* }$^+$ *f*←

Any service whose provision is prioritized may not be subject also to either synchronization or concurrency. Provision can, however, always be shared.

Note that prioritization may not be direct (the result of a single immediate prioritized alternation) but may follow from compound structure within the component.

Recommendations for Honeysuckle Programming Environment (HPE)

- An HVDL editor should offer the ability both to abstract a component, by depicting only its interface, and to *expand* it by progressively revealing more of its internal composition (prioritized service architecture).

5 System implementation

5.1 Value

5.1.1 Terminology

An *object* is something on which a *value* is written.

An object (variable) or value (constant) may be *named* within the context of a process.

An object is first named, then explicitly *created*, and simultaneously *assigned* a value. Creation can be delayed until an initial value may be determined.

Every object always has a value, though that value may be repeatedly reassigned (rewritten). An object must be read at least once after being written.

Every object is *owned* by precisely one process at any time. The owner may explicitly *destroy* it at any time, though destruction is automatic upon owner termination.

A value may be scalar or vector. A literal vector may be expressed as a *string* or *list*.

A *static value* is one which may be calculated once, before the program runs. A *dynamic value* is one free to vary as the program runs, which implies that it depends upon at least one object. (An object is unnecessary when assigned just once, but is nonetheless allowed for the sake of program development). A static value is independent of any object.

Independent of any object, a value may be named, irrespective of whether static or dynamic.

A *class* defines a set of values, together with any *operations* and *constraint* applicable.

Only one class is employed in Honeysuckle syntax – the set of numbers termed NATURAL.

5.1.2 Simple class

Enumeration

A *simple class* may be expressed as an *enumeration*:

```
definition of class WorkDay

  note
    ©2005 Ian East

  class WorkDay is
    Monday
    or Tuesday
```


©2006-10, Dr. Ian Robert East

```
or Wednesday
or Thursday
or Friday
```

Each element of an enumeration is called a CONSTANT.

‡*class.construction* ⟶ { *exposition* … }

‡ *exposition* ⟶ `is` { … | { ↲
→c *enumeration* … *f*← } }

enumeration ⟶ *element* { ↲c `or` *element* }⁺

element ⟶ *constant* …

constant ⟶ … *name*

Every class possesses a common constant in NULL. This may be awarded a more appropriate alias within each class – 0, in the case of NATURAL.

In an enumeration, NULL is identified with the first of any constants listed.

A Boolean decision, for example, to terminate repetition or select a process, is made according to whether the value of an expression is NULL or not. Thus an appropriate alias for NULL in a Boolean class definition would be FALSE.

Constant names are taken from the *universal* name-space, and thus cannot be redefined within any single program.

Semantic rule: A universal name-space is maintained against all composition.

Implementation note: Upon importing any process definition, the uniqueness of each element in the universal (and any other) name-space must be verified.

A constant may also be defined by a literal, when a character or number.

constant ⟶ *literal* | *name*

Constants can be shared between class definitions, which may thus overlap.

HPE requirement: Programmer must be warned of any overlap between class definitions, as these may be formed by an unintended reuse of a name.

Note that one or more *generic literal* constants may be employed as an implicit enumeration.

Predefined enumerations

HSE requirement: The following class enumerations are to be predefined:

- NATURAL
- INTEGER
- REAL
- CHARACTER.

Note that each of the above has a corresponding literal lexicon.

Equivalence

A class may instead be defined as an *equivalence* with an existing one:

```
class Length is a Natural
```

allowing, for example, integer length and integer time to be properly distinguished, while sharing a common set of values.

```
class Period is a Natural
```

‡ *exposition* ⟶ `is` { *citation* | { ↲
→c *enumeration* … f← } }

citation ⟶ { `a` | `an` } *range*

range ⟶ *name* …

<u>Semantic rule</u>: The choice between `a` and `an` is to be made correctly.

A citation may be subject to a *qualification*:

```
class HourCount is a Natural from 0 to 24
```

range ⟶ *name* [*subrange*]

<u>Semantic rule</u>: No object reference is permitted in a class citation subrange.

<u>Semantic rule</u>: A subrange may only be applied to an enumeration or an equivalence with one, and never a union or a *composition* (see below).

subrange ⟶ `from` *value* { { `to` *value* } | { `for` *value* } }

<u>Semantic rule</u>: The domain of a subrange must be *finite* and lie within a class that is *scalar*.

A subrange FROM 0 TO 24 could instead be indicated by FROM 0 FOR 25. This time, the second value need not be an explicit member of the *base class*. It merely counts out the number of legitimate values therein that will belong to the derived one.

For example:

```
class TrainingDay is a WorkDay from Monday for 2
```

<u>Semantic rule</u>: The value following TO must share the same class as that following FROM.

<u>Semantic rule</u>: The value following FOR must be of class NATURAL.

Because values are ascribed an order when listed, *any* simple class may be presumed ordinal.

(Recall that Honeysuckle defines a small set of data types whose literal values are recognized at the lexical level. This includes NATURAL and (signed) INTEGER.)

Union

A class may be defined as a *union*:

element ⟶ *constant* | *citation* …

For example:

```
class WeekDay is
   a WorkDay
```

 ©2006-10, Dr. Ian Robert East

```
      or a RestDay
```

Care is required with a union because operator applicability can vary within its domain. For this reason, processing *must* be subject to selection according to subordinate type:

```
if
  day is
    a WorkDay
      ...
    a RestDay
      ...
```

Semantic rule: Processing of any union must be subject to reduction by selection to basic classes or compositions.

5.1.3 Value

Introduction

A value may be expressed directly as a constant (member of a simple (enumerated) class), or recovered from an object:

‡ *value* ⟶ *constant* | *object* | …

‡ *object* ⟶ *name* | { `the` *name* } | …

The definite article ('the') may on occasion be employed to refer to a specific object. This merely has the effect of rendering phrasing more natural.

Scalar or vector

A value may be either *scalar* or *vector*. Honeysuckle draws no distinction between single and compound values, any more than it does between single or compound processes.

‡ *value* ⟶ *constant* | *object* | *string* | *list* | …

‡ *object* ⟶ *name* | { `the` *name* } | { *object* *index* }

‡ *index* ⟶ `[` { *value* | *subrange* } `]`

list ⟶ *h.list* | *v.list*

h.list ⟶ `(` *space* [*value* { `,` *space* *value* }*] *space* `)`

v.list ⟶ `(` ↲ →c *value* { ↲c *value* }$^+$ *space* `)` *f*← ↲

The object in question may be an element, or a *subrange*, of another.

There are, however, occasions when Honeysuckle requires a value to be scalar, such as when employed as an index, selecting a single element from a list, string, or array (see below).

A list describes a vector, even if it contains less than two elements. The notation can is used in assignment of both value and parameter list, and may be horizontal or vertical:

```
assign shopping ( sugar, salt )
```

©2006-10, Dr. Ian Robert East

or

```
assign shopping (
  sugar
  salt )
```

To be arranged vertically, a list must comprise more than one value.

Semantic note: An index value must be scalar.

Parsing note: Above expansions of *object* and *value* require LL(1) conflict resolution. They are retained in the given form for clarity here.

Expression

A value may also be the result of composition within an expression or a list:

‡ *value* ⟶ *constant* | *object* | *string* | *list* | *expression* | …

‡ *expression* ⟶ *prefix* | *infix*

‡ *prefix* ⟶ *p.operation value*

‡ *infix* ⟶ (*value i.operation value*)

i.operation ⟶ *b.operator* | *relation* | *name*

p.operation ⟶ *u.operator*

Except for the application of a prefix operator, value composition always requires the use of parenthesis. This is in contrast with Pascal and occam, which call for brackets only when a value is the subject of an infix operator or function.

Naming and reference

A value may be *named*. Once named, it may be subject to reference within the definition of some other name.

‡ *value* ⟶ *constant* | *object* | *string* | *list* | *expression* | *v.reference*

‡ *v.reference* ⟶ *name* …

There are two distinct, but semantically equivalent, ways in which a value may be named – as part of the *context* of a process, or as a separately-defined item:

```
{
  ...
  define
    String value greeting is \
    \ "Hello world!\n"
  ...
}
```

```
definition of String value greeting

  String value greeting is \
  \ "Hello world!\n"
```

The frequent need to name specific values warrants a direct facility within process definition. On the other hand, it may sometimes be preferable to define them as an item within a project.

©2006-10, Dr. Ian Robert East

Every named value forms an attribute of a process, class, or service. Even though it may be defined as an independent *item*, that definition must still be imported. An exception to this is when only the value itself is required (an intrinsic application).

Semantic rule: Every value naming must lie within the context of a process or class.

HPE requirement: Facility to review the hierarchical dependence of any dynamic value.

HPE requirement: *Interactive* evaluation of any defined value.
A *verbose* mode should display all necessary prior evaluation.

Every value definition implies the importation of its parent class. This may be *implicit* if it is a character, string, or number (*i.e.* its class is predefined).

Dynamic evaluation

A *dynamic* value is one that ultimately depends on an object. (At the atomic level, only the value written on an object is free to vary as the program runs.)

Class consistency requires:

Semantic rule: Within an expression, operator and value must belong to a common class. Any such class may be the result *promotion* or *unification*.

A value is considered dynamic if it is compound and any element is itself dynamic. (A list may combine values of different class, including scalar and vector.)

Conditional evaluation

An *evaluation* may depend upon a value from the applicable context. Such a value is termed a *parameter*. Evaluation is said to be *conditional* upon it. For example:

```
Natural value factorial of Natural x is
  ...
```

A single parameter can be given in the heading. Two or more must be listed below:

```
List value composition
  of
    List old
    List new
  is
    ...
```

Listing a value as a parameter, upon which evaluation does *not* depend, constitutes an error.

Reference to each parameter is by a name, defined only within the value definition. Note that such a name denotes *only* a value, not an object.

‡ *value.defn* ⟶ *name* `value` *name* { { [`of` *declaration*] `is` *evaluation* } |
{ ↲ → `of` ↲ →*c* *declaration* { ↲*c* *declaration* }$^{+}$ *f*← ↲ }
`is` *evaluation* ← }

declaration ⟶ *name* *name*

Invoking conditional evaluation implies passing an ‘actual’ parameter (list):

‡ *v.reference* ⟶ *name* [{ `of` *list* } | *list*]

<u>Semantic rule:</u> Any LIST cited in a value reference must not be empty.

No side-effect (update of the value of any external object) is possible. However, subsequent dynamic evaluation, satisfying a different reference, may well be affected.

Conditional evaluation is allowed only within a value naming (definition) – in-line or off-line – and not upon reference.

It may be necessary to describe *how* evaluation is undertaken, including perhaps the use of selection and recursion:

```
Natural value factorial of Natural x is
  if x is
    0, 1
      1
    otherwise
      factorial (x - 1)
```

This is termed *conditional evaluation*, and may only be employed when naming a value.

Conditional evaluation, like any other, may be either static or dynamic, according to the nature of each actual parameter passed.

No context can be declared within any evaluation.

‡ *evaluation* ⟶ *value* | { ↲ → *v.selection* ← }

‡ *v.selection* ⟶ `if` ↲ →c *v.sn.clause* { ↲c *v.sn.clause* }$^{+}$ *f*←

‡ *v.sn.clause* ⟶ { *v.sn.guard* ↲ → *value* ← } …

‡ *v.sn.guard* ⟶ *condition* | `otherwise`

condition ⟶ `is` *value*

A comment may precede any clause.

NO and NOT are synonyms for prefix negation:

<u>Semantic rule</u>: A condition fails when its result is NULL, but succeeds otherwise.

<u>Semantic rule</u>: Any recursion must terminate. (This may not be fully verified but direct self-reference must be denied.)

As well as applying a single condition, a guard may be *compound* and select according to the value of an expression:

‡ *v.sn.clause* ⟶ { *v.sn.guard* ↲ → *value* ← } | *v.cases* …

‡ *v.cases* ⟶ *subject* `is` { { *set* ↲ → *value* ← } | *v.sn.list* }

‡ *v.sn.list* ⟶ { ↲ → *set* ↲ → *value* ← }$^{2+}$ ←

subject ⟶ *value*

set ⟶ *list* | *subrange* | *citation*

<u>Semantic rule</u>: The *subject* of a *cases* clause must be scalar and dynamically evaluated.

 ©2006-10, Dr. Ian Robert East

Lastly, a compound guard may be formed via *replication*:

‡ *v.sn.clause* ⟶ { *v.sn.guard* ↲ → *value* ← } | *v.cases* | *v.sn.replication*

‡ *v.sn.replication* ⟶ `for any` *range* ↲ → *v.sn.clause* ←

Conditional evaluation has a strong similarity to *selection* in process construction. See #5.2.3 for a full explanation of its syntax.

Infix operator definition

To be completed.

Summary

Implementation note: Every value has three attributes according to whether it is:
- scalar or vector (simple or compound)
- static or dynamically evaluated
- an attribute (element in the context) of a process, object, or class.

5.1.4 Composite class

Record

A *composite class* is a *product* of others, typically termed a *record* – the analogue of a sequence within process or service. For example:

```
class Employee has
  a StaffCode
  and a Rank
```

Composition is an alternative to *exposition* in constructing a class:

‡*class.construction* ⟶ { *exposition* | *c.composition* }

‡ *c.composition* ⟶ `has` ↲
→*c* { … *product* } *f*←

‡ *product* ⟶ *component* { ↲*c* `and` *component* }$^{+}$

‡ *component* ⟶ … { *citation* *name* }

Note that AND replaces OR as a separator.

The NULL of a record is a list comprising the null of each component.

Reference to a field within a record is by field *class* name:

```
assign newEmployee[Rank] : CoPilot
```

Unlike Pascal, individual fields within a record do not have individual names. As a result, each field must have a distinct type. This is not as restrictive as it sounds, but it does require

type differentiation. Recall that Honeysuckle types are *name*-, not structure-, equivalent. For example, a name and an address might each be a STRING, yet still be distinct. The frequent need for type differentiation within a record is facilitated.

For example:

```
class Employee has
  a StaffCode
  and a String Name
  and a String Address
  and a Rank
```

The two new types – NAME and ADDRESS – are available wherever EMPLOYEE is.

Every composition must be created *complete*. For example:

```
create newEmployee : ( "Jones", "28 OX1 H33", Lacky )
```

In Honeysuckle, no record may *ever* be incomplete.

In any list, reference can be made only to *value*, and not to any process that might change state. It is thus not possible to assign any field within a record "on the fly".

Array

An array, in Honeysuckle, is a matter of *replication* of fields within a record.

‡ *c.composition* ⟶ `has` ↲
→*c* { *array* | *product* } ←

‡ *array* ⟶ `for each` *range* ↲ →*c* { *array* | *product* } *f*←

‡ *component* ⟶ *array* | { *citation* *name* }

For example:

```
class TimeSheet has
  for each WorkDay
    for each Commitment
      an HourCount
```

Predefining ranges facilitates matching repetition within process and value structure. A more traditional style is afforded ordinal ranges:

```
class FixedStringIndex is Natural from 1 for 64

class FixedString has
  for each FixedStringIndex
    a Character
```

The Honeysuckle System Environment (HSE) will offer prior definition of the classes NATURAL and CHARACTER, among others.

<u>Semantic rule</u>: Any class prefixed by EACH must be statically defined (without recursion or dynamically controlled replication).

©2006-10, Dr. Ian Robert East

There is no need to introduce ancillary indices in order to refer to any given field. An array element may be selected via the appropriate constant:

```
assign thisWeek[HourCount][Monday][DefeatMekon] : 18
```

assuming one commitment is the defeat of the dreaded green fellow with the big head.

A composite class that varies in size is expressed via *dynamic replication*:

```
class String has
  a Length
  and for each Natural from 1 for the Length
    a Character
```

Semantic rule: Within a record, any object referred to within the definition of one field must appear earlier as another, and the reference suffer the indefinite article.

A string literal, such `"beware the jabberwock"` is such a *dynamic array*.

A dynamically replicating class definition is self-contained, with no dependence upon context.

While dynamic memory allocation will be necessary in implementation, type-compatibility can be verified statically.

Union with record

An object may not always appear composite. It may sometimes possess a specific symbolic value. Recall that *any* class includes the constant NULL, even a composite one. (Including NULL in an enumeration has no effect.) Enumerating other specific constants, or providing an alias for null, can often be useful, and lead to more readable code.

‡ *exposition* ⟶ `is` { *citation* | { ↲
→*c* *enumeration* [`or` *c.composition*] *f*← } }

For example:

```
class Let is
  Vacant
  or has
    Date Start
    and Date End
    ...
```

Care is required with processing instances of any such class. Selection *must* be employed, and offer specific processing for every alternative.

Creation may often employ the ‘null’ constant, rather than the record, as in:

```
create newLet : Vacant
```

General union

Each constant within an enumeration is an example of a *generator*.

The HAS-notation, introduced to define record and array compositions is just "short-hand" for the introduction of *parameters* to one or more generators.

‡ *element* ⟶ *constant* | *citation* | { *name* `of` *product* }

The two definitions are *precisely* identical, including the use of the name COMPOSITION, which may be tested within a selection.

HAS-notation is limited to a single parameterized generator (COMPOSITION), whereas the more general syntax permits union with a number of distinct compositions (records).

See the following sub-section on recursion for further explanation and examples.

Recursion

Honeysuckle allows the definition of a class via recursion, and thus a *recursive data type*, using a notation similar to that proposed by Hoare [12].

Just as an explicit reference to a position within the text of a program is rejected, so is one to a location within memory. There is neither GOTO nor pointer ('reference') in Honeysuckle. There is thus no need of any automated "garbage collection" that would introduce significant and unpredictable latency at unpredictable times.

Recursion may be introduced via one or more parameters. For example:

```
class List is
  Empty
  or has
    an Item
    and a List
```

```
class List is
  Empty
  or composition of
    an Item
    and a List
```

These two definitions are identical.

A LIST may be created by either generator:

```
create shoppingList : Empty
```

or

```
create shoppingList : composition of [cheese, Empty]
```

Recall that every reference to any instance of any union *must* be via selection.

Multiple dimensions are possible, as are multiple parameterized generators:

```
class Family is
  an Individual
  or descendent of
    a Family and
    a Family
```

```
class EventTree is
  Empty
  or leaf of
    an Event
  or branch of
    an Event
    and an EventTree
    and an EventTree
```

Each of the above recursive definitions constitutes a *word algebra*. Recursion creates an infinite set, which rarely relates to reality. Hoare uses the term *class* to refer to a recursion

 ©2006-10, Dr. Ian Robert East

constrained by both an *invariant* condition and a set of applicable procedures and functions, which offer the only means by which the state of any instance may be changed.

To complete a Honeysuckle class definition, a *constraint* (class invariant) may be imposed by composition with *applicable routines*. Proof that a given set of routines does indeed impose the intended constraint is the duty of the designer.

It may be assumed within applicable routines that structure mirrors the order in which elements are combined via AND.

See the following section, on defining *abstract class* for syntax and example.

Genericity

Either replication or recursion (below) may incorporate a single *type parameter*, affording genericity over component type. For example:

```
class String of item has
  a Length
  and for each Natural from 1 for Length
    an item
```

An *actual* type parameter is provided upon naming, *not* upon creation:

```
named String of Integer : scoreList
```

All classes thus remain *statically* defined.

5.1.5 Abstract class

Applicable operations

To be defined

Constraint (invariant) assertion

Constraints act upon every generated value using applicable operations.

The effect of a broken constraint depends upon the HSE.

To be defined.

Inheritance

To be defined.

Generic operations

Generic operations are applicable to every class. Nine are recognized:

–	equality	=	infix
–	inequality	≠	infix

– identity IS infix
– existence IS prefix
– negation NOT prefix
– comparison $< \leq \geq >$ infix.

All return NULL upon failure, and TRUE with success.

Infix IS may be used either in place of '=', to test identity, *or* to test class membership:

```
day is Monday                 day is a WeekDay

is not raining
```

Prefix IS fails if NULL follows but succeeds otherwise.

5.1.6 Time

Time is an abstract class, supplied by the Honeysuckle System Environment (HSE).

Any time referred to within a block must be declared as part of its context.

```
named Time : receipt
```

The point at which some event occurs is then recorded:

```
sequence
  ...
  mark receipt
  ...
  if
    after 2 days from receipt
      ...
    ...
```

If merely a delay is required, with no alternative behaviour, a simpler expression is possible:

```
sequence
  ...
  mark receipt
  ...
  wait until after 2 days from receipt
```

MARK assigns the current time to an object of class Time.

Reference to a clock is implicit only, and is provided by the commands MARK and WAIT UNTIL, and by the applicable operators AFTER and BEFORE.

Time in Honeysuckle is *relative*, not absolute. Calendar events must be externally derived.

The domain of class Time is that of a 64-bit natural number, counting nanoseconds. It therefore exhibits a dynamic range of approximately 585 years with nanosecond resolution.

For convenience, class Time defines a variety of familiar units – nanoseconds, microseconds, milliseconds, seconds, minutes, hours, days, months, years, and centuries.

Both AFTER and BEFORE properly account for any "wrapping around" of the built-in clock.

 ©2006-10, Dr. Ian Robert East

AFTER and BEFORE do not merely return a Boolean result. They also *block progress* until either they or some alternative holds. They may thus form *criteria* for:

- selection *via* IF
- terminating repetition *via* REPEAT
- pre-emption (interruption) of one process by another *via* WHEN.

Criteria are said to *guard* clauses in a construction and are discussed within the description of the selection construction, IF.

Temporal criteria are expressed using the same syntax as any other applicable operation, but are incorporated directly within the Honeysuckle language, as one kind of *criterion*:

‡ *criterion* ⟶ *condition* | *timing* | *communication*

‡ *timing* ⟶ { `after` | `before` } *value* [`from` *name*]

In outline:

`after` *delay* `from` *mark*

If reference to a mark is omitted, it is replaced by one implicit in the previous command (WAIT UNTIL, IF, REPEAT, OR WHEN).

Implementation note: A *mark* is implemented as timed rendezvous. Thus every object of class Time has an associated rendezvous location, as does each instance of the command WAIT UNTIL, IF, REPEAT, OR WHEN.

The syntax of MARK and WAIT UNTIL is given within the description of *commands*.

5.2 Process

5.2.1 Categories of process

A process in Honeysuckle is expressed as one of the following:

- *command* primitive process
- *composition* of subordinate processes, either in sequence or in parallel defines a *component* or *system*
- *construction* sequence, repetition, selection, or alternation
- *recursion* a reference to one being defined
- *reference* to a process separately named and defined (offline).

process ⟶ *command* | *composition* | *construction* | *recursion* | *p.reference*

The purpose of a *composition* is to allow the reader to restrict concern to a single component at any time. That of a *construction* is to render an algorithm apparent. Structure thus reduces first via (de)composition, into components, and second via construction, into commands.

5.2.2 Commands

Overview

There is a total of eleven commands in four categories, according to their effect:

‡ *command* ⟶ *nihilistic* | *objective* | *temporal* | *communication* | *reactive*

nihilistic ⟶ `skip` | `stop`

objective ⟶ *create* | *destroy* | *assign*

‡ *temporal* ⟶ *mark* | *wait*

communication ⟶ *transfer* | *acquire* | *send* | *receive*

reactive ⟶ *disable* | `terminate`

Only nihilistic commands leave state unaffected, though STOP represents deadlock.

The following entities exist for reference within a direct command:

value name	name only, value to be received
object name	name only, object to be created or acquired
value	value defined or received
object	object that exists and is owned
port	one or other end of a service, within the declared interface.

These are distinguishable semantically but not syntactically. They appear all alike as a *name* within any production.

There follows a brief summary of each command:

skip	do nothing then terminate
stop	do nothing but never terminate
create	… an object with a specified initial value
destroy	… an object
assign	… an object a new value
transfer	… an object to a client/server
acquire	… an object from a client/server
send	… a value to a client/server
receive	… a value from a client/server
mark	… an object with the time
wait until	… a *temporal condition* is fulfilled
disable	… a guard event, or guard events, of an alternation
terminate	… an alternation (equivalent to disabling *all* its guard events).

Examples of direct commands:

```
create holiday
assign holiday Christmas Day
transfer order to accounts
acquire order from sales
send order to dispatch
receive order from sales
destroy order
```


©2006-10, Dr. Ian Robert East

```
mark receipt
wait until after receipt plus (2 times day)
disable time-out
terminate
```

Each complete command describes a primitive process. Primitive processes are primitive components with an *implicit interface*.

Any object reference may include a *subrange*.

Skip and stop

SKIP starts and immediately terminates.

STOP starts but never terminates. It represents *deadlock*.

Create, assign, destroy

As an example of an implicit interface, CREATE takes an object name and returns an object:

`create` *object name* :

ASSIGN *borrows* an object – rather as a Pascal procedure takes a “reference parameter”. In contrast, DESTROY *consumes* an object, returning nothing.

Semantic rule: The object to which an assignment refers must already exist.

create ⟶ `create` *name* :

destroy ⟶ `destroy` *name*

assign ⟶ `assign` { *object* | `this` } *value*

CREATE must always be followed by either ASSIGN or RECEIVE, without repeating reference to the object concerned:

```
...                                  ...
create holiday :                     create holiday :
assign this Christmas Day            receive this from line manager
...                                  ...
```

Semantic rule: Every object must be assigned value immediately after creation.

Some (static) protection is afforded against unintended reassignment.

Semantic rule: No object may be re-assigned before it may have been read.

An assignment is equivalent to the destruction of an object, followed by its re-creation:

```
sequence
  destroy object                    =       assign object value
  create object name :
  assign value
```

Only a complete object may be created, destroyed, or assigned (immediately after creation), whereas an element or subrange may be *re-assigned* independent of the remainder.

Transfer, acquire, send, and receive

Transfer of an object conveys *ownership* to the process that acquires it. Objects are thus *mobile* between processes.

`transfer` *object* `to` *port*

`acquire` *object name* `from` *port*

Object and value are distinguished by syntax rather than by explicit notation. There is no need for either pointer or 'reference'. The name following *send* refers to the *value* of an object, not the object itself. Any valid expression (means of determining a value) may follow.

`send` *value* `to` *port*

`receive` *value name* `from` *port*

In summary, three predicates are verified upon translation – declaration, existence plus ownership, and type-consistency. Attempting the assignment or transfer of any object that *might* not exist indicates either poor design or poor expression of algorithm and is thus denied.

‡ *mark* ⟶ `mark` *name*

‡ *wait* ⟶ `wait until` *timing*

transfer ⟶ `transfer` *name* `to` *name*

acquire ⟶ `acquire` *name* `from` *name*

send ⟶ `send` *value* `to` *name*

receive ⟶ `receive { signal |` *object* `| this } from` *name*

disable ⟶ `disable { this | {` *name* `{ ,` *name* `}*` `} }`

Only ASSIGN and RECEIVE, may refer to an element or subrange of an object, as well as the entire thing. TRANSFER and ACQUIRE must refer to an entire object, though SEND may be applied to an element or subrange.

Recall that a value also may reduce to an object, and note that a combination of SEND and RECEIVE has precisely the same effect as an assignment.

Mark, wait

See Section 5.1.D, re programming time-dependent behaviour.

Disable, terminate

See description of *alternation* in subsequent section on commands.

5.2.3 Process composition

Introduction

A composition (or *block*) combines processes, in either parallel or sequence.

©2006-10, Dr. Ian Robert East

Both *system* and *component* are described via hierarchical composition. The term 'system' merely refers to a composition not a component of any other ("higher level") composition.

Components composed in sequence communicate by inscribing values on shared *objects*; those in parallel communicate by synchronously passing values or objects according to a *service* protocol.

composition ⟶ { ↲
→c *interface values* ↲
{ { *state* ↲ *sequence* } | { *network* ↲ *parallel* } }
f← ↲
}

The *network* of service connections, or the declaration (naming) of objects that comprise *state*, must be declared in advance and forms part of the *context* of the composition:

```
{
  named
    ... objects

  sequence
    ... components
}
```

```
{
  network
    ... services

  parallel
    ... components
}
```

Note that object declaration (naming) occurs only above a sequential composition, and network declaration only above a parallel composition.

There is always a single blank line either side of a context declaration. (Recall, Honeysuckle is relentlessly strict regarding layout and format. There is no arbitrary "white space".)

An object named within a given context must be either *acquired* via a communication or explicitly *created* within the applicable scope.

Object naming (declaration) is straightforward:

state ⟶ `named` { *declaration* | { ↲
→c *declaration* { ↲c *declaration* }$^{+}$ f← } } ↲

as is the *construction* of a sequence:

sequence ⟶ `sequence` ↲ →c *process* { ↲c *process* }$^{+}$ f←

While a sequence construction may arise independently, only a sequential *composition* offers the opportunity to declare (name) objects and values. This is similar to C, where an embedded sequence (block) may introduce new variables.

Caution! Braces do *not* indicate the start and end of a sequence, as they do in C.

Services are declared only within a parallel composition, *i.e.* above a parallel construction:

parallel ⟶ `parallel` ↲ →c *process* { ↲c *process* }$^{+}$ f←

Network declaration is sufficient to document the design of a system or component, and may stand alone without implementation (parallel construction). It is also more complicated syntactically than the context for a sequential composition, and will thus be addressed separately.

Semantic rule: A naming may be followed only by a sequence, and a network by a parallel.

HPE requirement: A warning should be issued when a named object is assigned only once.

Process context and inheritance

Each and every reference (name) employed within a process must be declared in advance. Collectively, the set of references made make up the *process context*, which subsumes the context of composition (objects or services named).

Curly brackets (braces) circumscribe the *scope* of a given context.

```
{
  ... context

  ... composition
}
```

```
{
  ... outer context

  ...
  {
    ... inner context

    ...
  }
  ...
}
```

Elements of the context of an embedded block (child) may be *inherited*[1] from that of the outer one (parent). *Inheritance is not automatic*, as it is in Pascal or C. *Every* reference within any composition must be explicitly named, including any inheritance.

References inherited, or *returned* to a parent form the *sequential interface* of the process.

In addition, the context may extend to a *parallel interface*, and an *evaluation* (naming values).

A sequential composition may inherit values and objects *through* any parent parallel composition from a grandparent sequential composition.

Any *reference* (or 'invocation', see below) to a process composed in parallel may well pass parameters, which are received by its components. Each value passed may be received by multiple such components, but each object by precisely one.

---

[1] To emphasize the distinction, this may be referred to as *context inheritance*, and not to the *definition inheritance*, where one class 'inherits' attributes of another, and thus 'extends' its definition. The former usage predates the latter and originates with E. W. Dijkstra. Unlike the latter, it refers to the visibility of values or loan of objects already in use. However, there are clear similarities that suggest an alternate designation – *inline inheritance* versus *offline inheritance*.

©2006-10, Dr. Ian Robert East

In-line and off-line process definition

Because *every* reference must be declared, every block is self-contained and may be defined either *in-line* or *off-line* (like a Pascal procedure) with identical semantics:

```
process ... :                          process ... is
{                                        {
  // in-line definition                    // off-line definition

  ... context                              ... context

  ... composition                          ... composition
}                                        }
```

The only syntactic difference between an in-line and an off-line definition is associated with the preceding label which serves to name the process. Note that with an in-line definition there is no ensuing inset, except that within the block delimiters (curly brackets).

In-line definition is provided solely to support recursion (see #5.2.3), and infers a single instance of a process, albeit recurrent. External reference is disbarred. In contrast, an off-line (item) definition introduces a *class* of process, with the potential for multiple instantiation.

While an *alias* is syntactically an option, it is semantically obligatory for an in-line block, which has no other means of linkage. When the block is defined off-line, the "calling sequence" (in which parameters are listed upon invocation) establishes linkage. Giving an alias when defining a block off-line is a semantic error.

Semantic rule: No alias may be given in an off-line process definition (item).

Sequential interface

A sequential interface declares all references inherited or returned to a parent block.

interface ⟶ [*seq.interface*] …

A value may be *received* from the parent – perhaps derived from objects it owns. An existing object may be *borrowed*, and its value perhaps altered while on loan.

An object may also be *returned*, having been created locally.

seq.interface ⟶ [*receipts*] [*loans*] [*returns*]

The sequential interface of a process thus comprises up to three parts, and is stated first:

```
{
  received
    ... values
  borrowed
    ... objects
  returned
    ... objects
```

A sequential interface is very similar to a parameter list in Pascal. A borrowed object is nothing more than a "reference parameter", and a received value a "value parameter". Unlike Pascal, however, with Honeysuckle a value received does *not* become variable.

receipts ⟶ `received` { *s.i.declaration* | { ↲

→c *s.i.declaration* { ↲c *s.i.declaration* }⁺ f← } } ↲

loans ⟶ `borrowed` { *s.i.declaration* | { ↲
→c *s.i.declaration* { ↲c *s.i.declaration* }⁺ f← } } ↲

s.i.declaration ⟶ *declaration* [*alias*]

alias ⟶ `alias` *name*

Only the creator, not the borrower, of an object may destroy it. Loans must be returned.

Semantic rule: Any object borrowed must be assigned. Any value received must be used.

If not assigned then either no reference is made or only read access is required, in which case *value capture* is sufficient. Honeysuckle denies superfluous elements in an interface.

Every parameter is attributed an *alias* – an alternate name by which it is to be known locally. In an in-line definition, the relationship to the original (parental) name is rendered explicit. For example:

```
{
  named Day : holiday

  sequence
    create holiday : Christmas Day
    {
      received Day day off alias holiday

      ...
```

No alias need be added when the process is defined off-line, as it will be implicit in any *reference* (invocation). (The equivalent of a Pascal "actual parameter" must be given.)

Semantic rule: No ALIAS phrase may appear in the sequential interface of an off-line process definition (item).

A return must be named in the parental context, either as a declared object or again as a return. On inheritance, it may be given an alias, and must then be created or acquired.

returns ⟶ `returned` { *s.i.declaration* | { ↲
→c *s.i.declaration* { ↲c *s.i.declaration* }⁺ f← } } ↲

One important distinction exists in the invocation of a process according to whether it is defined in-line or off-line. When invoking one defined off-line, it is possible to compute a value passed immediately beforehand. A compound expression may be included in the process reference. No such opportunity exists when the process concerned lies in-line.

In its pursuit of transparency, Honeysuckle requires a strict separation between the delineation of any (necessarily simple) values passed from the context of a parent to that of its in-line offspring and the definition of any values computed from them. For example:

```
{
  received Natural y alias x

  define Natural value z is (2 × y)
```

 ©2006-10, Dr. Ian Robert East

. . .

It remains possible, though not essential, to supply an alias for a value passed.

If it lies within a parallel composition, a sequence can borrow an object, or receive a value, directly from its *grand-parent*. Its parent need not declare either; the grandchild can inherit directly. This preserves symmetry, in that only services need be declared over a parallel, and only objects over a sequence.

Parallel (component) interface

Any process composed in parallel with others requires declaration of its parallel interface:

interface ⟶ [*seq.interface* ↲] [*par.interface* ↲]

which documents the *ports* that connect it to the remainder of the declared network.

Ports are never individually named, but are referred to via the service (type or name) concerned, together with a *client* or *provider* attribute.

par.interface ⟶ `interface` ↲ →*c* … *f*← ↲

Honeysuckle regards system decomposition as primarily via division into parallel processes which communicate via synchronous channels under service protocol. As a result, the term ‘component’ is identified with ‘process’, and its parallel interface as the *component interface*.

Furthermore, in the Honeysuckle vocabulary, a *system* is totally self-contained and complete. It therefore possesses *no* interface.

Honeysuckle identifies ‘specification’ as the expression of each component interface. This may include some that are ‘virtual’ in that they represent behaviour that cannot be controlled, or programmed. These make up the world ‘external’ to Honeysuckle.

Design, in Honeysuckle, is the recursive reduction of a system into separate components. The language in which they and their *dependency*, is expressed is subordinate to Honeysuckle. It has both textual and visual counterparts – *Honeysuckle Design Language* and *Honeysuckle Visual Design Language* (HDL, HVDL) – that are documented below in Section 5.4. Design is expressed entirely in terms of service provision and consumption. Each component is identified solely as an *interface* between services, or as a *terminal* that either only provides or only consumes services.

In summary, each component is specified by its INTERFACE definition. The design of system and component is described by their NETWORK declaration (see below). Both specification and design are expressed using HDL or HVDL.

Implementation requires the definition of a process that fulfils the design of system or component. Even here, while processes are described, channels are not. They are the sole concern of the compiler, which must create at most two to implement any service.

Defining values (constants)

All values named within a composition must be defined within the context declaration:

values ⟶ `define` { *equation* | { ↲

→c *equation* { ↲c *equation* }$^{+}$ *f*← } } ↲

‡ *equation* ⟶ *name* `value` *name* { { [`of` *declaration*] `as` *evaluation* } |
{ ↲ → `of` ↲ →c *declaration* { ↲c *declaration* }$^{+}$ *f*← ↲ }
`as` *evaluation* ← }

A list of values can be defined below the heading DEFINE. For example:

```
{
  define
    Byte value write as #01
    Byte value read as #02

  ...
}
```

Recall that values may be either statically or dynamically evaluated. In the latter case, the name serves as an abbreviation for an expression incorporating an object reference. Evaluation may also be attributed 'structure' and parameters.

Only an element of syntactic sugar (AS versus IS) differentiates the 'off-line' definition of a value (as a separate item) from one 'inline' (within the context declaration of a process).

Furthermore, any value may be *conditionally* evaluated (see #5.1.3).

Network declaration

The final element in the context of a process is the *context of composition* – either the naming of objects shared within a sequential composition, or a *network declaration* defining the structure of a parallel one. It effects a *design* of component or system.

A network is declared using a subset of Honeysuckle – the *Honeysuckle Design Language* – which is documented in Section 4 above.

5.2.4 Process construction

Summary

A process may be *constructed* by combining others in sequence or according to repetition, selection, or prioritized alternation:

construction ⟶ *sequence* | *repetition* | *p.selection* | *alternation*

Note that only a *composition* may be parallel, while processes may be either composed *or* constructed in sequence.

Sequence

We have already met sequential construction of a process within a *composition*:

sequence ⟶ `sequence` ↲ →c *process* { ↲c *process* }$^{+}$ *f*←

 ©2006-10, Dr. Ian Robert East

It is permissible to *construct* a sequence *without* rendering context explicit. Instead, context is *implicit* and confined to a subset of the context of the parent process.

No new objects may be named, nor values defined, in *any* construction.

Selection

A single construction affords selection between processes according to a variety of criteria:

criterion ⟶ *condition* | *timing* | *communication*

Each process is said to be *guarded.*

p.sn.guard ⟶ *criterion* | `otherwise`

Guards are *ranked* according to the order in which they appear (highest priority uppermost).

Only one guard may succeed. The consequently selected process executes exactly once.

p.selection ⟶ `if` ↲ →*c* *p.sn.clause* { ↲*c* *p.sn.clause* }$^+$ *f*←

p.sn.clause ⟶ { *p.sn.guard* ↲ →*c* *process* *f*← } | *p.cases* | *p.sn.replication*

A single means of expression is employed for all selection:

```
if
  is raining
    ...
  otherwise
    ...
```

```
if
  day is
    Monday
      ...
    Tuesday
      ...
```

```
if
  receive order from Command
    ...
  after 2 × day
    ...
```

Timing and communication criteria are capable of *blocking* progress, which will *resume* via the first guard to become *ready*, as with the occam ALT construct.

As with occam IF/ALT, the process with the first successful (true/ready) guard, from the top of the list down, will be chosen.

Because the order in which guards are listed is significant, the effect of a *pre-condition* upon any communication or temporal criterion can be easily achieved. One need only enter the inverse earlier.

OTHERWISE is a guard that always succeeds – the equivalent of IS TRUE.

Any second OTHERWISE clause would be redundant, and is considered a semantic error.

Semantic rule: IF clauses must be distinct in their guard, though they may share domain.

Semantic rule: Unless a domain shared between clauses is accounted for in its entirety – which may or may not be verified, according to implementation – an otherwise clause is mandatory. Any Boolean domain must be so accounted.

To effect the equivalent of Pascal IF, reliance is placed upon the generic prefix IS operator. This succeeds with any operand other than NULL (0, in the case of any numeric class).

Although, as a generic operator, prefix IS is already defined, its expectation here requires incorporation within selection syntax:

condition ⟶ `is` *value*

Semantic rule: A condition fails when its result is NULL, but succeeds otherwise.

NO and NOT are synonyms for prefix negation:

```
if                          if
  is no workDone              is not raining
    ...                         ...
  otherwise                   otherwise
    ...                         ...
```

Selection according to the value of an expression is expressed using the *infix* IS operator, which tests either identity or inclusion within a class.

Again, this must be reincorporated to achieve the desired syntax:

p.cases ⟶ *value* `is` { *p.set.item* | *p.sn.list* }

p.set.item ⟶ *set* ↲ →c *process* f←

p.sn.list ⟶ ↲ →c *p.set.item* { ↲c *p.set.item* }$^{+}$ f←

set ⟶ *list* | *subrange* | *citation*

Semantic rule: The subject of a CASES clause must be scalar and dynamically evaluated.

A single conditional criterion may be expressed this way, or a list. If the list fails to exhaust the applicable domain, an additional (perhaps OTHERWISE) clause will be required.

There are two ways of testing whether a value lies between two bounds:

```
if
  character is from '0' to '9'
    ...
```

```
if
  day is
    a WeekDay from Monday for 2
     ...
```

Each is more elegant, transparent, and concise, than the usual Boolean combination of relational expressions.

A time criterion can be achieved via the prefix operators AFTER and BEFORE which will block until they succeed, or some alternative succeeds.

timing ⟶ { `after` | `before` } *value* [`from` *name*]

In outline:

`after` *delay* [`from` *mark*]

Any point of selection (start of an IF process) forms an *implicit mark* from which a delay can begin. If no reference is made to a explicit mark, the previous implicit one applies.

A delay is of class *natural* – a natural integer.

Communication criteria are expressed in precisely the same manner as commands. Refer to documentation of:

– *send* – *receive* – *transfer* – *acquire*

Honeysuckle guarantees the absence of deadlock with one exception; STOP may be explicitly programmed, perhaps for purposes of testing a response to failure.

 ©2006-10, Dr. Ian Robert East

Unlike occam, which allows only receipt to be the criterion for selection, Honeysuckle allows either input- or output-guard. To avoid the need for negotiation [14], a selection guard at both input *and* output end of a communication is disbarred.

Semantic rule: No communication may be selective at both ends.

By ‘selective’, we mean that communication is definitely *offered* but will not occur unless the partner so wishes. This is not to be confused with a communication being *subject* to selection according to some explicit criterion. Communication would then be referred to as *conditional*.

While a communication may not be selective at both ends, it may be conditional at both ends, provided they share a *common* condition. It may also be conditional at one end and selective at the other. In fact, this is a very useful combination, affording an “if something happens then tell me” protocol. For example, it is how *exception handling* is performed, although selective response within an alternation is perhaps more likely than within selection.

Replication is permitted, over elements of some object. Control will pass via the successful guard with the lowest index. For example, if WORKDONE is an array over domain WORKDAY:

```
if
  for any WorkDay
    is no workDone
      send WorkDay to ...
  otherwise
    skip
```

The reference to WORKDAY in the conditionally executed process will recover the element in its domain for which the guard succeeded. (Remember a constant is not processed text.)

A new keyword ANY is introduced, in place of EACH, to avoid the suggestion of repetition. The name that follows refers to a class over whose domain the object ranges. A class subrange may also be added as a qualifier.

p.sn.replication ⟶ `for any` *range* ↲ →c *p.sn.clause* f←

Note that the index (here, a working day) is not cited in the criterion. Reference may be made directly to an object whose elements range according to the class (subrange) cited above.

Every value *must* be accounted for within a CASE list *unless* a communication, temporal, or OTHERWISE clause, offers an alternative. The lack of any (eventually) successful guard would result in deadlock – the equivalent of STOP, which starts but never terminates.

Semantic rule: Every selection must have an opportunity for success, via either communication, finite time delay, or elaboration of an entire domain.

Implementation note: The internal representation of class must include a record of the size of the applicable domain, or an indication that it is indefinite.

In practice, this rule leaves three possibilities:

- elaboration of an entire domain within a single *cases* clause
- inclusion of one or more timing or communication guards

- inclusion of a final OTHERWISE clause.

Note that elaboration of *cases*, as originally proposed by Hoare as an extension to Algol [15], and typified by the Pascal CASE construction, is arguably more fundamental than selection according to a single condition (Pascal IF). It is simple to elaborate a logical domain.

Honeysuckle combines Pascal IF and CASE in a single construction, along with occam ALT.

No sensible justification can be made for object creation within a selection. Any object required in any particular clause can be the property of a block subtended there. Conditional creation would require subsequent conditional use, inviting disaster, and preventing the necessary static (compile-time) guarantee that every reference is to an object which exists.

There remains the possibility that the initial value of an object requires selection, even though creation does not. Some might then prefer to express creation + initialization in every clause. However, a single creation, followed by a selection between substitute 'initial' values is just as logical, and retains a simple rule – objects are never conditionally created.

Repetition

As with selection, all forms of repetition may be expressed using a single construction:

```
repeat
  while ...
    ...
```

```
repeat
  always
    ...
  while ...
```

```
repeat
  for each ...
    ...
```

In addition to the equivalent of Pascal WHILE-DO, REPEAT-UNTIL (DO-WHILE, in *C*), and FOR constructions, Honeysuckle affords the additional *prime* form identified by Maddux [11], sometimes called DO-WHILE-DO:

```
repeat
  always
    ... do this
  while ...
    ... do that
```

Under the heading REPEAT, a sequence of processes is given, each guarded by a condition. Repetition terminates immediately upon failure of *any* guard. ALWAYS merely a short-hand for WHILE TRUE. A missing process is interpreted as SKIP – do nothing, then terminate.

repetition ⟶ `repeat` ↲ →c *r.body* f←

r.body ⟶ *r.sequence* …

r.sequence ⟶ *r.clause* { ↲c *r.clause* }*

r.clause ⟶ *r.guard* [↲ →c *process* f←]

r.guard ⟶ `always` | { `while` *value* }

Sequential execution of clauses is implicit, as it is of within REPEAT-UNTIL in Pascal.

Semantic rule: At least one clause within a repetition must contain a process.

Objects may not be created within a repetition. No context is declared. Every repetition is thus subordinate to (embedded within) a sequence.

 ©2006-10, Dr. Ian Robert East

A comment may precede each clause.

Indexed repetition may be recursively be introduced via FOR EACH. All indices are subsequently implicit and must not be cited. A sub-range may be added, as in selection.

r.body ⟶ *r.sequence* | *r.replication*

r.replication ⟶ `for each` *range* ↲ →c *r.body* f←

Semantic rule: To simplify translation, each value in any replication range must be static.

Implementation note: No replication index may ever exceed the given range, which infers a guarantee of remaining within its domain. (This is not the case in some other languages, where an index must *exceed* a stated range before a condition fails and repetition terminates.)

Replication merely introduces an additional exit. Within its scope, a list of conditions may continue to provide exits, as before. An embedded condition will, in other words, exit the *entire* repetition, and not just the current iteration. Selection will be necessary to effect behaviour that varies with iteration; for example, to skip one.

Alternation

Honeysuckle provides direct expression of *prioritized alternation* between processes, guarded by timing or communication *events*. At least two processes must be listed.

```
when
  receive ...
    ... do this
  after ...
    ... do that
  ...
  quiet
    ... background tasks
```

alternation ⟶ `when` ↲ →c *a.clause* { ↲c *a.clause* }* f←

Alternation may be considered a third form of process (component) *composition*, after sequence and parallel. It may also be considered a sequential construction, akin to selection.

Guards are attributed distinct *relative* priority, and are listed accordingly, from high to low.

Upon starting, each guard event is automatically *enabled*. It may then proceed to become *ready*. The corresponding *response* will normally follow immediately. Event and response effectively form a *component* via sequential composition.

When multiple guards are simultaneously ready, the one with the highest priority will occur.

Any response is subject to *pre-emption* by another guard, of higher priority. Execution of the interrupted process resumes once the response to the higher priority event is *complete*.

Each component is cyclic about its guard. No guard event may recur until the completion of a response to a previous occurrence, *i.e.* no component of an alternation is *re-entrant*.

Such behaviour is more commonly known as (non re-entrant) "prioritized vectored interruption" and is supported by the architecture of most processors, virtual and physical. As a result, response *latency* is typically a few clock cycles, at most.

QUIET represents a guard that is always ready, and semantically identical to SKIP. Just one such clause may be included. It must be listed last, with lowest priority.

An alternation remains a sequential process. No two components ever execute concurrently. However, it must be used with great care, as the each interruption is equivalent to a GOTO. The mapping from procedure to process is necessarily one of high dimension.

It is strongly recommended that a design evolve first without alternation, and its correctness (correspondence with specification) verified. Only when confidence may be placed in uninterruptible processes should they be composed in alternation.

Every guard remains enabled until explicitly *disabled* via the DISABLE command, which may only be issued within an alternation. DISABLE THIS disables the parent guard. TERMINATE disables all guards. Alternation will then terminate once all initiated responses are complete.

An alternation always terminates once all guards, including QUIET, are disabled.

Any guard may be disabled in any response. Each may be given a name that may serve as a reference. For example:

```
when
  answering
    receive ...
      ... respond
  time_out
    after ...
      sequence
        ... tidy up
        terminate
  house-keeping
    quiet
      ... background tasks
```

$$
\begin{aligned}
\textit{a.clause} &\longrightarrow \textit{a.named.clause} \mid \textit{a.unnamed.clause} \\
\textit{a.named.clause} &\longrightarrow \textit{name} \;\hookleftarrow\; \rightarrow c \;\; \textit{a.unnamed.clause} \;\; f\leftarrow \\
\textit{a.unnamed.clause} &\longrightarrow \{ \;\; \textit{a.guard} \;\hookleftarrow\; \rightarrow c \;\; \textit{process} \;\; f\leftarrow \;\; \} \mid \textit{a.replication} \\
\textit{a.guard} &\longrightarrow \textit{communication} \mid \textit{timing} \mid \texttt{quiet}
\end{aligned}
$$

It is also possible to simultaneously disable a list of guards:

```
disable [one, two, three]
```

A consequence of the semantics of alternation is that the result of nesting one construction inside another is that the result is similar, but not equivalent, to a single construction alone. The only possible difference is a delay in enabling an embedded clause if its parent alternation follows additional commands in sequence.

©2006-10, Dr. Ian Robert East

Any QUIET clause of a nested alternation would deny execution of any such clause in the parent. Honeysuckle seeks to remove the expression of impossible behaviour as misleading.

Semantic rule: Only one QUIET clause is allowed within nested alternation constructions.

Even the possibility of non-compliance with an interface is denied, and must be rejected by a compiler. For example, this denies alternation at both ends of a communication, just as selection at both ends is denied.

Semantic rule: All communication must accord with an applicable declared interface.

It is strongly recommended that communication guards each denote the initiation of a service. The response may then continue it to completion. Partial rendition of a service by a component of an alternation invites error.

Objects may not be created within an alternation. No private context is declared. Every alternation is thus subordinate to (embedded within) a sequential or parallel composition.

A common context is shared between all components of an alternation. Though visible within any component, no object may be assigned value within more than one. A component changing the state of an object is known as its *keeper*.

Semantic rule: An object in the context of an alternation may possess at most one keeper.

Prioritized alternation may be used to effect response to *exceptions* – events that indicate an error has occurred. Expressing a response this way is appropriate where there are many opportunities for some error, and where the alternative of frequent selection would "clog up" a program, making it difficult to both write and read.

Rectification of, or recovery from, errors dynamically (at "run-time") is rarely possible. In practice, any response will usually reduce to the following:

- ignore
- reset (restart)
- terminate.

The difficulty lies in satisfying all concurrent processes that anticipate communication. To deny resumption of an interrupted errant component could be fatal to the wider system. Furthermore, it may be the wider system that needs to be reset or terminated. A secure algorithm exists to accomplish this [16], but requires the entire system to be designed accordingly.

A useful design pattern is to form a *check-point* at WHEN by embedding it within first a sequence and then a loop. An exception response might effect reset by terminating the alternation via TERMINATE, whereupon its state might be returned to a standard configuration prior to re-entry.

An 'ignore' response can be achieved simply via SKIP, and one of termination – of the entire alternation – via TERMINATE. An additional dedicated command is included to effect reset.

An exception remains an external signal, rather like a neighbour raising an alarm when the spot a fire in your house. The Honeysuckle System Environment (HSE) is expected to include a *watchman* that provides a basic suite of exception signals, including at least the following:

- arithmetic overflow (an operation causes domain violation)
- domain violation (illegal array index, value outside the domain of a class)

– attempt to divide by zero.

Watchman protocol is documented as an illustration of service definition in Section 5.3, and as part of the requirements for the HSE in Appendix C.

As with repetition and selection, an alternation may employ replication. Response to an entire array of events may thus be expressed with just a single extra line. For example:

```
when
  for any Registered
    receive request from Client
      ... respond
  quiet
    ... keep house
```

where 'Client' is an array of identical server ports.

Replication may be nested to any depth.

a.replication ⟶ `for any` *range* ↲ →c *a.unnamed.clause* *f*←

Semantic rule: To simplify translation, each value in any range must be static.

5.2.4 Recursion

A *recursion* is simply a process defined in terms of itself. In other words, the definition contains at least one *reference* (see below) to itself.

The term may instead refer to each such reference.

For example, a procedure to create a new document in, say, a word processor might include the means by which a user can create a further document:

```
...
process present.new.document :
{
  ... context

  ...
    ...
      present.new.document
}
...
```

An inline definition is simultaneous with command issue (*e.g.* 'present.new.document!').

Note the option to abstract a document as a process, as an alternative to an object. Note also that a name is chosen accordingly, to express a command.

For any recursion to terminate, a parameter must be passed. Hence, the process defined can only be a sequential composition.

Semantic rule: A recursive composition may be sequential only, never parallel.

©2006-10, Dr. Ian Robert East

Syntax is a little more specific than that of an offline definition, within an *item*. (See Section 4.1.)

recursion ⟶ `process` *name* **:** ↲ *composition*

Recursion is the *sole* purpose served by an in-line definition.

Semantic rule: Both naming without recursion and any external reference are disallowed.

An off-line definition must be provided to satisfy multiple references to the same process.

Recursion is also permitted within an off-line definition.

5.2.5 Reference

Invocation

A *reference* to a process constitutes an invocation of it, with the option to pass parameters.

A reference may occur within either kind of composition, sequential or parallel, or any kind of construction.

A reference may form part of an in-line definition, in which case the reference constitutes a recursion, or be to a process defined off-line (as an *item*).

Three distinct parameter lists may be given, corresponding to the sequential, parallel, and network, interface of the process invoked.

Honeysuckle favours extending code downwards, rather than across. This will often remove the need for line continuation. Experience also suggests that "loo-roll" code is more readable.

Sequential interface

Parameters are listed, in the order in which they appear in the process context. For example:

```
...
encrypt message, encryption
...
```

```
process encrypt is
  {
    received
      String plain
    returned
      String encrypted

    ...
```

Here, the actual parameter MESSAGE refers to an object whose value is passed. Encryption refers to an object, named by the caller, but created by the procedure.

An alternative is to *lend* an object for the procedure to update:

```
...
encrypt message
...
```

```
process encrypt :
{
  borrowed
    String message
  ...
```

(It is assumed we are happy to overwrite the plain message with the encrypted version.)

Parameters may be listed either horizontally, comma-separated, or vertically, indented.

p.reference ⟶ *name parameters*

parameters ⟶ *list* …

Self-reference, within a recursion, takes an identical form.

Semantic rule: Every external process reference must be to an off-line (item) definition.

A reference is recognized within either composition or construction by the leading name, that must refer to a process either defined or being defined:

Multiple parameters must be listed beneath the reference.

Parallel and configuration interface

Reference may be to a process which is a component in a parallel composition. Such a process may have an additional *parallel interface.*

Actual services to be provided or consumed may be passed as *configuration parameters.*

Configuration values are a second kind of configuration parameter, and may be passed in a third parameter list. The three distinct parameter lists are delimited by semi-colon.

‡ *parameters* ⟶ *list* [; *list* [; *list*]]

Semantic rule: No recursive reference may include configuration parameters.

Semantic rule: Configuration values must be of type NATURAL.

Any empty list preceding one that is not empty invites an empty parameter list field. For example:

```
parallel
  ...
  mediate ; s2 ; 4
  ...
```

```
process mediate is
  {
    ... no sequential interface
    interface
      client of s1 alias ?
      ...

    network
      received
        Length
      ...
```

In the above example, a single service name is passed, along with one configuration value, perhaps the length of a chain of subordinate processes.

©2006-10, Dr. Ian Robert East

5.3 Service

5.3.1 Simple service

Mimicking an occam channel

A service must be defined, as a distinct item, before use:

‡ *service* ⟶ *s.communication* …

s.communication ⟶ *s.transfer* | *s.acquire* | *s.send* | *s.receive*

s.transfer ⟶ `transfer` { *object* | *citation* }

s.acquire ⟶ `acquire` { *object* | *citation* }

s.send ⟶ `send` { *value* | *citation* }

s.receive ⟶ `receive` { … | *object* | *citation* }

Note that communication within the definition of a service differs from that within a process. Rather than refer to a specific object, we may simply cite its class.

For example, a "hello world" process requires a simple console on which to display a greeting:

```
definition of service console

  imports
    class String from StandardTypes

  service Console is
    receive a String
```

```
definition of process greet

  imports
    service Console

  process greet is
    {
      interface
        client of Console
      ...
```

At its simplest, we may just require the receipt of a STRING (an instance of class STRING). With no further reference required, no name need be attributed. The needs of a service definition are here less than that of any corresponding process.

A service is defined from the perspective of the *provider* (server), not the client.

Simple service accomplishes no more than a *channel* definition in occam. It merely restricts the type of value that may be transmitted, and attributes an orientation for data-flow. The only difference is that, unlike occam, Honeysuckle allows the transfer of an object, and not just the copying of a value.

Any system that can be defined in terms of channels can also be defined in terms of service. Service protocol is able to place *additional* constraint upon behaviour, beyond that which may be expressed via channel abstraction, by restricting the order in which communications may occur. In so doing, it *raises the level of abstraction.*

Recall that item definition is transparent, in the sense that any item imported brings along with it any definitions it in turn imports. For example, in importing service console, process greet automatically acquires a definition of class STRING.

Syntax allows a service to require transmission of a constant. This is not expected to be terribly useful, but for one exception:

Signals

Where only synchronization is required, without communication of any message, Honeysuckle provides for the sending of a mere *signal*:

```
definition of service Sentinel

  service Sentinel is
    send signal
```

Recall that, in a service definition, nothing is said about the processing that might precede any communication, which might include a delay. The signal might be a "wake-up" call, days (or even years) after the provider starts. This is perfectly consistent with the requirements for a *service network component* (SNC) [10]. It must be live, but it is allowed to doze.

A service may also, at some point, await a mere signal from the client:

s.receive ⟶ `receive` { `signal` | *object* | *citation* }

5.3.2 Compound service

Sequence

Service definition extends the power of a programmer to express behaviour chiefly by constraining the order in which communications can occur.

‡ *service* ⟶ *s.communication* | *s.construction* …

‡ *s.construction* ⟶ *s.sequence* | …

s.sequence ⟶ `sequence` ↲ →c *service* { ↲c *service* }$^{+}$ *f*←

Handshaking is a simple, commonplace, and useful, example:

```
definition of service Console

  imports
    class String from StandardTypes

  service Console is
    sequence
      receive a String
      send a String
```

Any process pair whose interface comprises console service are now compelled to exchange strings "tit-for-tat". Channel protocol cannot express this. To see how communication is ordered, we would be forced to descend to the implementation. Honeysuckle allows both a more abstract interface to be expressed and a guarantee of its implementation.

 ©2006-10, Dr. Ian Robert East

Selection

Recall that a service is a protocol governing the order in which communications may occur. That order remains free to vary, subject to mutual agreement. Variation may be signalled by either party, and indicated by a value communicated or by choice between alternative communications at any point within service conduct:

‡ *s.construction* ⟶ *s.sequence* | *s.selection* | …

‡ *s.selection* ⟶ `if` ↲ →c *s.sn.clause* { ↲c *s.sn.clause* }$^+$ *f*←

‡ *s.sn.clause* ⟶ { *s.sn.guard* ↲ →c *service* *f*← } | *s.cases*

‡ *s.sn.guard* ⟶ *condition* | *s.communication* | `otherwise`

‡ *s.cases* ⟶ *value* `is` { *s.set.item* | *s.sn.list* }

‡ *s.set.item* ⟶ *set* ↲ →c *service* *f*←

‡ *s.sn.list* ⟶ ↲ →c *s.set.item* { ↲c *s.set.item* }$^+$ *f*←

For example, console service may vary according to the initial command received:

```
service Console is
  {
    define
      Byte write : #01
      Byte read  : #02
    named
      Byte command

    sequence
      receive command
      if
        command is
          write
            acquire a String
          read
            sequence
              receive a Natural
              transfer a String
  }
```

Expressing the required behaviour requires declaration of an object and definition of certain values. These are collectively termed the *interface context.*

‡ *service* ⟶ *s.communication* | *s.construction* | *s.composition* …

‡ *s.composition* ⟶ { ↲
→c *seq.interface* *values* ↲ *state* ↲ *s.sequence* *f*← ↲
}

The name attributed an object within a service definition is ignored in a process that implements it (either as client or provider). It merely allows identification between reference, and

so allows a decision to be taken accordingly. Client and provider must name, and create, their own objects.

Values remain defined and need not be redefined in any implementing process.

Although a service definition resembles that of a process, it amounts only to a *template*, governing the pattern of communication. As such it is an explicit constraint on the degree of non-determinacy allowed any (system or component) process.

As with process definition, service definition may nest sequential composition.

Selection may also be made according to the communication chosen by the client. Selection within service protocol merely establishes alternate branches. Recall that *no communication may be selective at both ends*. As a result, the provider *defers* to the client.

With the following definition, it might at first appear that payment will never be acquired and that service will always terminate after the dispatch of (a copy of) the invoice:

```
service Business is
  {
    ...

    sequence
      acquire an order
      send an invoice
      if
        acquire a payment
          transfer an item
        otherwise
          skip
  }
```

Such is not the case. Either payment is acquired, then an item transferred, or no further transaction between client and provider takes place. The definition simply endorses either as legitimate. Perhaps the business makes use of a timer service and decides according to elapsed time whether to accept or refuse payment if/when offered.

Repetition and recursion

A compound service may also be constructed via repetition. It might seem unnecessary, given that a service protocol is inherently repeatable anyway, but account must be taken of other associated structure.

‡ *s.construction* ⟶ *s.sequence* | *s.selection* | *s.repetition*

‡ *s.repetition* ⟶ `repeat` ↲ → *s.r.body* ←

‡ *s.r.body* ⟶ *s.r.replication* | *s.r.clause*$^+$

‡ *s.r.clause* ⟶ *r.guard* ↲ [→c *service* f←]

‡ *s.r.replication* ⟶ `for each` *range* ↲ →c *s.r.body* f←

Semantic rule: To simplify translation, each value in any range must be static.

 ©2006-10, Dr. Ian Robert East

For example, the following might be a useful protocol for copying each week between two diaries:

```
service Diary is
  {
    ...

    sequence
      repeat
        for each Week Day
          send a Day
      send a Week
  }
```

As with process exposition, greater clarity, or efficiency of expression, can often be achieved via recursion.

‡ *service* ⟶ *s.communication* | *s.construction* | *s.composition* | *reference*

For example:

```
service document menu is
  {
    ...

    if
      command is
        ...
        open
          sequence
            ...
            document menu
    ...
  }
```

As with a process reference, a service reference may refer to one or more parameters.

Reference may similarly be made to any other service item.

Unlike process definition, no embedded (inline) definition is allowed. All recursion must be to the current item.

©2006-10, Dr. Ian Robert East

Appendix A: Honeysuckle Programming Language (HPL-2009)

A.1 Character set

A.1.1 Program text

A Honeysuckle program, with the exception of literal characters and strings, is inscribed using only the following characters:

- digits `0–9`
- letters `a–z A–Z`
- brackets `( ) [ ] { }`
- punctuation `: ' " , . ?`
- symbols `# \`
- operators `+ − × ÷ \ / ^ ∧ ∨ ¬`
- relations `= ≠ < > ≤ ≥`
- format space NBS SNL

NBS refers to a "non-breaking space", which is used as an option to bind multiple words together to form a name.

SNL denotes a command to "start new line".

Both SNL and 'space' characters are elements of Honeysuckle syntax. They are *not* ignored as mere "white space" as they are in other languages.

While Honeysuckle is independent of any particular character encoding scheme, it is expected that this will be the 16-bit *Unicode Translation Format*, UTF-16.

A.1.2 Processed character set (PCS)

The PCS reduces to those characters whose UTF-16 encoding lies between:

- `#0020–007E` or `#00A0–0D7FF`

Three distinct categories of gremlin should be reported by the lexical analyst (scanner):

- an unsupported control code in `#0000–001F` `#007F–009F`
- an unsupported character in `#F900–FDCF` `#FDF0–FDFD`
- no character in `#D800–DFFF` `#E000–F8FF` `#FDD0–FDEF` `#FFFE–FFFF` .

The following table summarizes escape sequences recognized:

`\n`	line-feed (new line) NL	`\#`	#
`_`	non-breaking space NBS	`\\`	\

©2006-10, Dr. Ian Robert East

`\'`	`'`	`\"`	`"`
`#hhhh`	character whose encoding is $hhhh_{16}$ (h = hex digit)		

A.1.3 Character class definitions

space ⟶ #0020
SNL ⟶ #000A
NBS ⟶ #00A0

digit ⟶ `0–9`
hex.digit ⟶ *digit* | `A-F`
letter ⟶ `a–z` | `A–Z`

quote ⟶ `'` | `"`
escape ⟶ `\` | `#`
b.operator ⟶ + | − | × | ÷ | \ | / | ^ | ∧ | ∨
u.operator ⟶ + | − | ¬
relation ⟶ = | ≠ | < | > | ≤ | ≥
hanging ⟶ − | (

any ⟶ #0020–#007E | #00A0–#D7FF
any.in.quotes ⟶ *any* \ { *quote* | *escape* }
any.on.line ⟶ *any* \ SNL

A.2 Lexis

A.2.1 Keywords and key characters

Keywords

`a`	`acquire`	`after`	`alias`	`alternate`	`always`	`an`
`and`	`any`	`as`	`assign`	`before`	`borrowed`	`class`
`client`	`collection`	`create`	`define`	`defines`	`definition`	`destroy`
`disable`	`each`	`for`	`from`	`has`	`if`	`imports`
`in`	`interface`	`is`	`mark`	`named`	`network`	`not`
`null`	`of`	`or`	`otherwise`	`parallel`	`process`	`provider`
`quiet`	`receive`	`received`	`repeat`	`returned`	`send`	`sequence`

©2006-10, Dr. Ian Robert East

`service`	`signal`	`skip`	`stop`	`terminate`	`the`	`then`
`this`	`to`	`transfer`	`until`	`value`	`wait`	`when`
`while`						

Keywords are *reserved* and cannot thus be confused with *names*.

Key characters

`>`	`?`	`,`	`;`	`:`	`>`	`?`
`(`	`)`	`[`	`]`	`{`	`}`	`\`
+	−	×	÷	\	/	^
=	≠	<	>	≤	≥	
∧	∨	¬	space			

A.2.2 Format commands

Verified

$$\textit{return'} \longrightarrow \textit{full.return} \mid \{ \ \textit{short.return} \ \ \textit{hanging} \ \}$$

$$\textit{inset'} \longrightarrow \textit{full.indent} \mid \{ \ \textit{short.indent} \ \ \textit{hanging} \ \}$$

$$\textit{full.return} \longrightarrow \text{SNL} \ \ \textit{indent}^{n}$$

$$\textit{full.indent} \longrightarrow \textit{space} \ \ \textit{space}$$

$$\textit{short.return} \longrightarrow \textit{indent}^{n-1} \ \ \textit{space}$$

$$\textit{short.indent} \longrightarrow \textit{space}$$

Reported (perceived by parser)

$$\hookleftarrow \ \longrightarrow \textit{return}$$

$$\rightarrow \ \longrightarrow \textit{inset}$$

$$\leftarrow \ \longrightarrow \textit{inset}^{-1}$$

$$\textit{continuation} \longrightarrow \textit{space} \ \ \backslash \ \ \hookleftarrow \ \ \backslash \ \ \textit{space}$$

$$\textit{return} \longrightarrow \textit{full.return} \mid \textit{short.return}$$

$$\textit{inset} \longrightarrow \textit{full.indent} \mid \textit{short.indent}$$

$$\rightarrow^{p} \ \leftarrow^{q} \ \longrightarrow \varepsilon \qquad (p = q)$$

©2006-10, Dr. Ian Robert East

$\rightarrow^{p} \leftarrow^{q} \longrightarrow \rightarrow^{p-q}$ $(p > q)$

$\rightarrow^{p} \leftarrow^{q} \longrightarrow \leftarrow^{q-p}$ $(p < q)$

Semantic rule: The degree of indentation *n* must be continuously recorded.

Semantic rule: Upon INSET, increment *n*.

Semantic rule: Upon OUTSET, verify $n > f$ (fold inset), then decrement *n*.

Implementation note: Lexical analysis must be capable of generating a sequence of tokens according to the constitution of a single lexeme.

Implementation note: An intermediate "lexical parser" might recognize both CONTINUATION and RETURN (↲) as these each comprise a sequence of tokens.

A.2.3 Literal values

literal ⟶ `signal` | `null` | *number* | *character*

string ⟶ `"` *c.character** `"`

number ⟶ *natural* | *integer* | *real*

character ⟶ `'` *c.character* `'`

natural ⟶ *digit*+ | { `#` *hex.digit*+ }

integer ⟶ [+ | -] *natural*

real ⟶ [+ | -] *digit*1+ [. *digit*+] [`E` [+ | -] *digit*+]

c.character ⟶ *any.in.quotes* | { *hex.quad* } |
{ `\` { `n` | `_` | `#` | `\` | `'` | `"` } }

hex.quad ⟶ `#` *hex.digit*4

Semantic rule: Recognition of any NUMBER is subject to the condition that its value lies within a domain of at least one category (NATURAL, INTEGER, or REAL). The lowest consistent category will be recorded.

Implementation note: Lexical analysis may consume, but need not report, TEXT.

Implementation note: While mapping to machine representation and verification of machine constraint satisfaction are logically the task of semantic analysis, it serves efficiency to conduct these tasks within lexical analysis.

©2006-10, Dr. Ian Robert East

A.2.4 Names

name ⟶ *letter* { [NBS | .] { *letter* | *digit* } }*

<u>Lexical rule</u>: No name may coincide with any keyword. (Keywords are ‘reserved’.)

A.2.5 Operators

i.operation ⟶ *b.operator* | *relation* | *name*

p.operation ⟶ *u.operator*

A.2.6 Annotation

Annotation is ideally removed within lexical analysis, which must then take responsibility for the enforcement of *the rules of folding* (see #A.3.7).

Lexis then ideally includes the following symbols, which are *not* reported to parser:

comment ⟶ { *comment.line* | *fold-in* | *comment.fold* } ↲

comment.line ⟶ // *space* *text*

fold.in ⟶ {{{ *space* *text*

fold.out ⟶ }}}

comment.fold ⟶ *comment.fold.in* ↲ *fold.out*

comment.fold.in ⟶ {{{// *space* *text* { ↲ *text* }*

text ⟶ *any.in.quotes**

A.3 Syntax

A.3.1 Item and collection

item ⟶ `definition of` *item.declaration* ↲
→c [*importation*] *definition* *f*← ↲

‡ *collection* ⟶ `collection` *name* ↲ ↲
→c [*importation*] [*manifest*]
definition { ↲ *definition* }*
f← ↲

©2006-10, Dr. Ian Robert East

item.declaration ⟶ *item.type name*

importation ⟶ ↲ `imports` ↲ →c { *import* { ↲c *import* }* f← ↲

‡ *manifest* ⟶ ↲ `defines` ↲
→c { *item.declaration* { ↲c *item.declaration* }+ f← ↲

definition ⟶ ↲ { *value.defn* | *class.defn* | *process.defn* | *service.defn* }

‡ *item.type* ⟶ `value` | `class` | `process` | `service`

‡ *import* ⟶ { *item.declaration* [`from` *name*] } |
{ `from` *name* ↲
→c { *item.declaration* { ↲c *item.declaration* }+ f← } |
{ `collection` *name* }
→c { *item.declaration* { ↲c *item.declaration* }+ f← ↲

‡ *value.defn* ⟶ *name* `value` *name* { { [`of` *declaration*] `is` *evaluation* } |
{ ↲ → `of` ↲ →c *declaration* { ↲c *declaration* }+ f← ↲ }
`is` *evaluation* ← }

‡ *class.defn* ⟶ `class` *name* [`of` *name*] *class.construction*

process.defn ⟶ `process` *name* `is` ↲ → *process* ←

service.defn ⟶ `service` *name* `is` ↲ → *service* ←

declaration ⟶ *name name*

Semantic Rule: The type of item defined must match that announced.

Semantic Rule: The list of definitions in a collection must match that listed in its manifest.

Semantic Rule: Importing an item or collection already imported is disallowed.

Implementation note: Any project must contain at least one ITEM or COLLECTION.

Implementation note: Syntactic analysis may always be completed without encountering end-of-file (EOF). Any such encounter (within lexical analysis) should thus be reported as an error. (A parser should cease requesting tokens from the scanner when an item or collection is complete.)

Implementation note: No ITEM or COLLECTION is complete until the fold-stack is empty.

A.3.2 Value

Evaluation

‡ *evaluation* ⟶ *value* | { ↲ → *v.selection* ← }

Intrinsic value

‡ *value* ⟶ *constant* | *object* | *string* | *list* | *expression* | *v.reference*

constant ⟶ *literal* | *name*

‡ *object* ⟶ *name* | { the *name* } | { *object* *index* }

list ⟶ *h.list* | *v.list*

‡ *expression* ⟶ *prefix* | *infix*

‡ *v.reference* ⟶ *name* [{ of *list* } | *list*]

‡ *index* ⟶ [{ *value* | *subrange* }]

h.list ⟶ (*space* [*value* { , *space* *value* }*] *space*)

v.list ⟶ (↲ →c *value* { ↲c *value* }+ *space*) f← ↲

‡ *prefix* ⟶ *p.operation* *value*

‡ *infix* ⟶ (*value* *i.operation* *value*)

subrange ⟶ from *value* { { to *value* } | { for *value* } }

Semantic note: An index value must be scalar.

Semantic rule: The domain of a *subrange* must be finite and lie within a class that is scalar.

Semantic rule: Any LIST cited in a value reference must not be empty.

Parsing note: Above expansions of *object* and *value* require LL(1) conflict resolution. They are retained in the given form for clarity here.

Conditional evaluation

‡ *v.selection* ⟶ if ↲ →c *v.sn.clause* { ↲c *v.sn.clause* }+ f←

‡ *v.sn.clause* ⟶ { *v.sn.guard* ↲ → *value* ← } | *v.cases* | *v.sn.replication*

‡ *v.sn.guard* ⟶ *condition* | otherwise

‡ *v.cases* ⟶ *subject* is { { *set* ↲ → *value* ← } | *v.sn.list* }

‡ *v.sn.replication* ⟶ for any *range* ↲ → *v.sn.clause* ←

condition ⟶ is *value*

 ©2006-10, Dr. Ian Robert East

A.3.3 Process

Kinds of process

process ⟶ *command* | *composition* | *construction* | *recursion* | *p.reference*

Primitive processes

‡ *command* ⟶ *nihilistic* | *objective* | *temporal* | *communication* | *reactive*

nihilistic ⟶ `skip` | `stop`

objective ⟶ *create* | *destroy* | *assign*

‡ *temporal* ⟶ *mark* | *wait*

communication ⟶ *transfer* | *acquire* | *send* | *receive*

reactive ⟶ *disable* | `terminate`

create ⟶ `create` *name* **:**

destroy ⟶ `destroy` *name*

assign ⟶ `assign` { *object* | `this` } *value*

‡ *mark* ⟶ `mark` *name*

‡ *wait* ⟶ `wait until` *timing*

transfer ⟶ `transfer` *name* `to` *name*

acquire ⟶ `acquire` *name* `from` *name*

send ⟶ `send` *value* `to` *name*

receive ⟶ `receive` { `signal` | *object* | `this` } `from` *name*

disable ⟶ `disable` { `this` | { *name* { **,** *name* }* } }

<u>Semantic rule</u>: The object to which an assignment refers must already exist.

<u>Semantic rule</u>: Every object must be assigned value immediately after creation.

Composition

composition ⟶ { ↲
→c *interface values* ↲
{ { *state* ↲ *sequence* } | { *network* ↲ *parallel* } }
f← ↲
}

©2006-10, Dr. Ian Robert East

sequence ⟶ `sequence` ↲ →c *process* { ↲c *process* }$^+$ *f*←

parallel ⟶ `parallel` ↲ →c *process* { ↲c *process* }$^+$ *f*←

Construction

construction ⟶ *sequence* | *repetition* | *p.selection* | *alternation*

repetition ⟶ `repeat` ↲ →c *r.body* *f*←

p.selection ⟶ `if` ↲ →c *p.sn.clause* { ↲c *p.sn.clause* }$^+$ *f*←

alternation ⟶ `when` ↲ →c *a.clause* { ↲c *a.clause* }* *f*←

p.sn.clause ⟶ { *p.sn.guard* ↲ →c *process* *f*← } | *p.cases* | *p.sn.replication*

p.sn.guard ⟶ *criterion* | `otherwise`

p.cases ⟶ *value* `is` { *p.set.item* | *p.sn.list* }

p.sn.replication ⟶ `for any` *range* ↲ →c *p.sn.clause* *f*←

r.body ⟶ *r.sequence* | *r.replication*

r.sequence ⟶ *r.clause* { ↲c *r.clause* }*

r.replication ⟶ `for each` *range* ↲ →c *r.body* *f*←

r.clause ⟶ *r.guard* [↲ →c *process* *f*←]

r.guard ⟶ `always` | { `while` *value* }

a.clause ⟶ *a.named.clause* | *a.unnamed.clause*

a.named.clause ⟶ *name* ↲ →c *a.unnamed.clause* *f*←

a.unnamed.clause ⟶ { *a.guard* ↲ →c *process* *f*← } | *a.replication*

a.replication ⟶ `for any` *range* ↲ →c *a.unnamed.clause* *f*←

a.guard ⟶ *communication* | *timing* | `quiet`

‡ *criterion* ⟶ *condition* | *timing* | *communication*

p.set.item ⟶ *set* ↲ →c *process* *f*←

p.sn.list ⟶ ↲ →c *p.set.item* { ↲c *p.set.item* }$^+$ *f*←

set ⟶ *list* | *subrange* | *citation*

condition ⟶ `is` *value*

‡ *timing* ⟶ { `after` | `before` } *value* [`from` *name*]

Semantic rule: IF clauses must be distinct in their guard, though they may share domain.

©2006-10, Dr. Ian Robert East

Semantic rule: Unless a domain shared between clauses is accounted for in its entirety – which may or may not be verified, according to implementation – an otherwise clause is mandatory. Any Boolean domain must be so accounted.

Semantic rule: A condition fails when its result is NULL, but succeeds otherwise.

Semantic rule: The subject of a CASES clause must be scalar and dynamically evaluated.

Semantic rule: No communication may be selective at both ends.

Semantic rule: Every selection must have an opportunity for success, via either communication, finite time delay, or elaboration of an entire domain.

Semantic rule: At least one clause within a repetition must contain a process.

Semantic rule: To simplify translation, each value in any replication range must be static.

Semantic rule: Only one QUIET clause is allowed within nested alternation constructions.

Semantic rule: All communication must accord with an applicable declared interface.

Semantic rule: An object in the context of an alternation may possess at most one keeper.

Semantic rule: To simplify translation, each value in any range must be static.

Implementation note: The internal representation of class must include a record of the size of the applicable domain, or an indication that it is indefinite.

Implementation note: No replication index may ever exceed the given range, which infers a guarantee of remaining within its domain. (This is not the case in some other languages, where an index must *exceed* a stated range before a condition fails and repetition terminates.)

Recursion and reference

recursion ⟶ `process` *name* : ↵ *composition*

p.reference ⟶ *name parameters*

‡ *parameters* ⟶ *list* [; *list* [; *list*]]

Semantic rule: A recursive composition may be sequential only, never parallel.

Semantic rule: Both naming without recursion and any external reference are disallowed.

Semantic rule: Every external process reference must be to an off-line (item) definition.

Semantic rule: No recursive reference may include configuration parameters.

Semantic rule: Configuration values must be of type NATURAL.

Context

interface ⟶ [*seq.interface* ↵] [*par.interface* ↵]

values ⟶ `define` { *equation* | { ↵

→c *equation* { ↲c *equation* }$^+$ f← } } ↲

state ⟶ `named` { *declaration* | { ↲
→c *declaration* { ↲c *declaration* }$^+$ f← } } ↲

seq.interface ⟶ [*receipts*] [*loans*] [*returns*]

‡ *equation* ⟶ *name* `value` *name* { { [`of` *declaration*] `as` *evaluation* } |
{ ↲ → `of` ↲ →c *declaration* { ↲c *declaration* }$^+$ f← ↲ }
`as` *evaluation* ← }

receipts ⟶ `received` { *s.i.declaration* | { ↲
→c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲

loans ⟶ `borrowed` { *s.i.declaration* | { ↲
→c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲

returns ⟶ `returned` { *s.i.declaration* | { ↲
→c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲

s.i.declaration ⟶ *declaration* [*alias*]

alias ⟶ `alias` *name*

PAR-INTERFACE and NETWORK are defined below.

Semantic rule: A naming may be followed only by a sequence, and a network by a parallel.

Semantic rule: Reference to any object created by a component of a parallel composition must be confined within that component. It must *not* occur in any other.

Semantic rule: Any object borrowed must be assigned. Any value received must be used.

Semantic rule: No ALIAS phrase may appear in the sequential interface of an off-line process definition (item).

HPE requirement: A warning should be issued when a named object is assigned only once.

Time

‡ *criterion* ⟶ *condition* | *timing* | *communication*

‡ *timing* ⟶ { `after` | `before` } *value* [`from` *name*]

Implementation note: A *mark* is implemented as timed rendezvous. Thus every object of class Time has an associated rendezvous location, as does each instance of the command WAIT UNTIL, IF, REPEAT, OR WHEN.

©2006-10, Dr. Ian Robert East

A.3.4 Network (Honeysuckle Design Language – HDL)

Interface

‡ *par.interface* ⟶ `interface` ↲ →c *p.i.element* { ↲c *p.i.element* }* f← ↲

‡ *p.i.element* ⟶ { *port* | *synchronization* | *prioritization* }

‡ *port* ⟶ { { `provider of` [*degree*] } |
{ `client of` } } *service.id*

‡ *synchronization* ⟶ `synchronized` { *degree provider* } | ↲
→c *n.provider* { ↲c *n.provider* }+ f←

‡ *prioritization* ⟶ `prioritized` { *degree provider* } | ↲
→c *n.provider* { ↲c *n.provider* }+ f←

service.id ⟶ *name* [`alias` { *name* | `?` }]

‡ *provider* ⟶ `provider of` *service.id*

‡ *n.provider* ⟶ [*degree*] *provider*

‡ *degree* ⟶ `[` *natural* `]`

Network

‡ *network* ⟶ `network` ↲
→c [*receipts*] [*state*] *n.element* { ↲c *n.element* }+ f← ↲

‡ *n.element* ⟶ { *provision* | *dependency* | *chain* | *selection* | *alternation* }

‡ *provision* ⟶ [{ `synchronized` | `distributed` } [*degree*]] *s.name*

dependency ⟶ *provision* `>` *consumption* { { `,` | `;` } *consumption* }*

‡ *chain* ⟶ `repeat for` *value* ↲
→c *dependency* { ↲c *dependency* }* f←

selection ⟶ `select` ↲ →c *n.s.element* { ↲c *n.s.element* }+ f←

‡ *alternation* ⟶ `alternate` [`for` *value*] ↲
→c *n.a.element* { ↲c *n.a.element* }+ f←

n.s.element ⟶ { *provision* | *dependency* }

‡ *n.a.element* ⟶ [`-`] { *provision* | *dependency* | *selection* }

‡ *consumption* ⟶ [`|`] *name* [`-`]

‡ *s.name* ⟶ *name* [`:` *natural* [`:` *natural*]]

©2006-10, Dr. Ian Robert East

Design rule: Any additional dependency of the lower-priority service must be arranged in sequence, and not interleaved, with the feedback path when fed forward to further interleaved provision, along with the higher-priority path. The order of the sequence is immaterial.

Design rule: But for services that are sequentially consumed, it must be possible to draw the service digraph so that no two dependencies cross.

Semantic rule: Any dependency may be expressed only once.

Semantic rule: No two alternating services may depend upon a shared service.

Semantic rule: Any feedback must ultimately depend upon a service provided at higher priority, *i.e.* the *feedback chain* must end at a point higher than its start. (This requires feedback to be indicated upon provision before consumption.)

Semantic rule: In any feedback chain, at least one dependency must be interstitial.

Semantic Rule: The degree quoted must exceed one.

Semantic rule: Where a service is consumed internally (entirely within a component), the number of providers must not exceed the number of consumers.

Semantic rule: Any degree is a natural number that must exceed one.

Implementation note: Parsing must include detection of services consumed within the same interstice within the service provided.

Implementation note: When the feedback chain consists of a single dependency (*i.e.* is of length one), a special method of code generation will be required, replacing each SEND/RECEIVE combination with assignment.

Implementation note: Clients of a synchronized service must be distinguishable.

Implementation note: Sharing requires a queue, rather like entry to a monitor, together with an additional rendezvous location to synchronize access.

Implementation note: Synchronized sharing requires a secondary queue from which elements are prevented from joining the primary one until a cycle is complete.

A.3.5 Class

‡*class.construction* ⟶ { *exposition* | *c.composition* }

‡ *exposition* ⟶ `is` { *citation* | { ↲
→c *enumeration* [`or` *c.composition*] *f*← } }

‡ *c.composition* ⟶ `has` ↲
→c { *array* | *product* } *f*←

citation ⟶ { `a` | `an` } *range*

enumeration ⟶ *element* { ↲c `or` *element* }$^+$

 ©2006-10, Dr. Ian Robert East

‡ *array* ⟶ `for each` *range* ↲ →*c* { *array* | *product* } *f*←

‡ *product* ⟶ *component* { ↲*c* `and` *component* }$^+$

‡ *component* ⟶ *array* | { *citation* *name* }

‡ *element* ⟶ *constant* | *citation* | { *name* `of` *product* }

constant ⟶ *literal* | *name*

range ⟶ *name* [*subrange*]

Semantic rule: The choice between `a` and `an` is to be made correctly.

Semantic rule: No object reference is permitted in a class citation subrange.

Semantic rule: A subrange may only be applied to an enumeration, or an equivalence with one, and never a union or a *composition* (see below).

Semantic rule: The domain of a subrange must be *finite* and lie within a class that is *scalar*.

Semantic rule: The value following TO must share the same class as that following FROM.

Semantic rule: The value following FOR must be of class NATURAL.

Semantic rule: Any class prefixed by EACH must be statically defined (without recursion or dynamically controlled replication).

Semantic rule: Within a record, any object referred to within the definition of one field must appear earlier as another, and the reference suffer the indefinite article.

Semantic rule: Processing of any union must be subject to reduction by selection to basic classes or compositions.

Semantic rule: A universal name-space is maintained against all composition.

Semantic rule: To simplify translation, each value in any range must be static.

Implementation note: Upon importing any process definition, the uniqueness of each element in the universal (and any other) name-space must be verified.

HPE requirement: Programmer must be warned of any overlap between class definitions, as these may be formed by an unintended reuse of a name.

HSE requirement: The following class enumerations are to be predefined:

- NATURAL
- INTEGER
- REAL
- CHARACTER.

Note that each of the above has a corresponding literal lexicon.

©2006-10, Dr. Ian Robert East

A.3.6 Service

‡ *service* ⟶ *s.communication* | *s.construction* | *s.composition* | *reference*

s.communication ⟶ *s.transfer* | *s.acquire* | *s.send* | *s.receive*

s.transfer ⟶ `transfer` { *object* | *citation* }

s.acquire ⟶ `acquire` { *object* | *citation* }

s.send ⟶ `send` { *value* | *citation* }

s.receive ⟶ `receive` { `signal` | *object* | *citation* }

‡ *s.construction* ⟶ *s.sequence* | *s.selection* | *s.repetition*

s.sequence ⟶ `sequence` ↲ →c *service* { ↲c *service* }$^+$ *f*←

‡ *s.selection* ⟶ `if` ↲ →c *s.sn.clause* { ↲c *s.sn.clause* }$^+$ *f*←

‡ *s.repetition* ⟶ `repeat` ↲ → *s.r.body* ←

‡ *s.sn.clause* ⟶ { *s.sn.guard* ↲ →c *service* *f*← } | *s.cases*

‡ *s.sn.guard* ⟶ *condition* | *s.communication* | `otherwise`

‡ *s.cases* ⟶ *value* `is` { *s.set.item* | *s.sn.list* }

‡ *s.set.item* ⟶ *set* ↲ →c *service* *f*←

‡ *s.sn.list* ⟶ ↲ →c *s.set.item* { ↲c *s.set.item* }$^+$ *f*←

‡ *s.r.body* ⟶ *s.r.replication* | *s.r.clause*$^+$

‡ *s.r.clause* ⟶ *r.guard* ↲ [→c *service* *f*←]

‡ *s.r.replication* ⟶ `for each` *range* ↲ →c *s.r.body* *f*←

‡ *s.composition* ⟶ { ↲
→c *seq.interface* *values* ↲ *state* ↲ *s.sequence* *f*← ↲
}

<u>Semantic rule</u>: To simplify translation, each value in any range must be static.

A.3.7 Annotation

↲c ⟶ ↲ [*comment* | *fold.out*]

→c ⟶ → [*comment*]

f← ⟶ [↲ *fold.out*] ←

©2006-10, Dr. Ian Robert East

comment ⟶ { *comment.line* | *fold-in* | *comment.fold* } ↲

comment.line ⟶ // *space text*

fold.in ⟶ {{{ *space text*

fold.out ⟶ }}}

comment.fold ⟶ *comment.fold.in* ↲ *fold.out*

comment.fold.in ⟶ {{{// *space text* { ↲ *text* }*

text ⟶ *any.in.quotes**

Semantic rule: A FOLD-OUT must follow an unmatched FOLD-IN.

Semantic rule: Matching FOLD-IN and FOLD-OUT must share a common inset – $n = f$.

Semantic rule: No line within a fold may possess an inset less than that of FOLD-IN/OUT.

Semantic rule: A compiler must record the degree of indentation (increment on each INSET, decrement on each OUTSET) and verify each rule above.

Implementation note: A scanner need report only the presence of a COMMENT, and not its internal structure.

Implementation note: Lexical analysis may report each COMMENT, as well as every INSET, OUTSET, and RETURN (→ ← ↲), but need not necessarily distinguish the type of comment (whether COMMENT-FOLD or COMMENT-LINE).

Implementation note Lexical analysis may report each FOLD-OUT, and distinguish each FOLD-IN, only if the parser is to enforce the above rules for folding. However, it is anticipated that this task may be performed within lexical analysis, in which case FOLD-OUT may be ignored (*i.e.* go unreported) and FOLD-IN reported simply as a COMMENT.

Implementation note: Lexical analysis must afford recognition of any form of COMMENT including any enfolded or terminating RETURN.

HPE requirement: An editor is required capable of toggling the status of any fold. It should be possible to accomplish this in as simple a manner as possible – ideally with a single mouse-click or key press.

HPE requirement: An editor should display the comment-status of a fold, using the line comment mark ('//'), whether open or closed:

– `... // when closed`

–
```
{{{ // when open
...
}}}
```

Note the space introduced between fold-in and comment marks.

Appendix B: Honeysuckle Core Language (HCL-2009)

B.1 Character set

The character set employed in HCL is precisely the same as in HPL. Refer to #A.1.

B.2 Lexis

B.2.1 Keywords

Keywords

Keywords in HCL are identical to those in HPL, but with certain omissions:

a	acquire		alias	alternate	always	an
	any	as	assign		borrowed	class
client	collection	create	define		definition	destroy
disable	each	for	from		if	imports
in	interface	is		named	network	not
null	of	or	otherwise	parallel	process	provider
quiet	receive	received	repeat	returned	send	sequence
service	signal	skip	stop	terminate	the	then
this	to	transfer		value		when
while			//	{{{//	{{{	}}}

Keywords are *reserved* and cannot thus be confused with *names*.

Key characters

Key characters in HCL are precisely the same as in HPL. Refer to #A.2.1.

B.2.2 Format commands

Format commands employed in HCL are precisely the same as in HPL. Refer to #A.2.2.

©2006-10, Dr. Ian Robert East

B.2.3 Literal values

Literal values employed in HCL are precisely the same as in HPL. Refer to #A.2.3.

B.2.4 Names

Names employed in HCL are precisely the same as in HPL. Refer to #A.2.4.

B.2.5 Operators

Operators employed in HCL are precisely the same as in HPL. Refer to #A.2.5.

B.2.6 Annotation

Annotation in HCL, as in HPL, may be considered either lexical, and removed by lexical analysis, or part of the syntax. Refer to #A.2.6.

B.3 Syntax

B.3.1 Item and collection

	item	⟶	`definition of` *item.declaration* ↲ →c [*importation*] *definition* f← ↲
	item.declaration	⟶	*item.type* *name*
	importation	⟶	↲ `imports` ↲ →c { *import* { ↲c *import* }* f← ↲
	definition	⟶	↲ { *class.defn* | *process.defn* | *service.defn* }
‡	*item.type*	⟶	`class` | `process` | `service`
‡	*import*	⟶	*item.declaration*
	definition	⟶	↲ { *value.defn* | *class.defn* | *process.defn* | *service.defn* }
‡	*value.defn*	⟶	*name* `value` *name* `is` *value*
‡	*class.defn*	⟶	`class` *name* *class.construction*
	process.defn	⟶	`process` *name* `is` ↲ → *process* ←
	service.defn	⟶	`service` *name* `is` ↲ → *service* ←

Semantic Rule: The type of item defined must match that announced.

Semantic Rule: Importing an item already imported is disallowed.

Implementation note: Any project must contain at least one ITEM.

Implementation note: Syntactic analysis may always be completed without encountering end-of-file (EOF). Any such encounter (within lexical analysis) should thus be reported as an error. (A parser should cease requesting tokens from the scanner when an item is complete.)

Implementation note: No ITEM is complete until the fold-stack is empty.

B.3.2 Value

‡ *value* ⟶ *constant* | *object* | *string* | *expression* | *name*

constant ⟶ *literal* | *name*

‡ *object* ⟶ *name* | { `the` *name* }

‡ *expression* ⟶ (*value* *i.operation* *value*)

list ⟶ *h.list* | *v.list*

h.list ⟶ (*space* [*value* { , *space* *value* }*] *space*)

v.list ⟶ (↲ →c *value* { ↲c *value* }+ *space*) *f*← ↲

B.3.3 Process

Kinds of process

process ⟶ *command* | *composition* | *construction* | *recursion* | *p.reference*

Primitive processes

‡ *command* ⟶ *nihilistic* | *objective* | *communication* | *reactive*

nihilistic ⟶ `skip` | `stop`

objective ⟶ *create* | *destroy* | *assign*

communication ⟶ *transfer* | *acquire* | *send* | *receive*

reactive ⟶ *disable* | `terminate`

create ⟶ `create` *name* :

destroy ⟶ `destroy` *name*

assign ⟶ `assign` { *object* | `this` } *value*

transfer ⟶ `transfer` *name* `to` *name*

©2006-10, Dr. Ian Robert East

acquire ⟶ `acquire` *name* `from` *name*

send ⟶ `send` *value* `to` *name*

receive ⟶ `receive` { `signal` | *object* | `this` } `from` *name*

disable ⟶ `disable` { `this` | { *name* { `,` *name* }* } }

Semantic rule: The object to which an assignment refers must already exist.

Semantic rule: Every object must be assigned value immediately after creation.

Composition

composition ⟶ `{` ↲
→c *interface* *values* ↲
{ { *state* ↲ *sequence* } | { *network* ↲ *parallel* } }
f← ↲
`}`

sequence ⟶ `sequence` ↲ →c *process* { ↲c *process* }+ f←

parallel ⟶ `parallel` ↲ →c *process* { ↲c *process* }+ f←

Construction

construction ⟶ *sequence* | *repetition* | *p.selection* | *alternation*

repetition ⟶ `repeat` ↲ →c *r.body* f←

p.selection ⟶ `if` ↲ →c *p.sn.clause* { ↲c *p.sn.clause* }+ f←

alternation ⟶ `when` ↲ →c *a.clause* { ↲c *a.clause* }* f←

p.sn.clause ⟶ { *p.sn.guard* ↲ →c *process* f← } | *p.cases* | *p.sn.replication*

p.sn.guard ⟶ *criterion* | `otherwise`

p.cases ⟶ *value* `is` { *p.set.item* | *p.sn.list* }

p.sn.replication ⟶ `for any` *range* ↲ →c *p.sn.clause* f←

r.body ⟶ *r.sequence* | *r.replication*

r.sequence ⟶ *r.clause* { ↲c *r.clause* }*

r.replication ⟶ `for each` *range* ↲ →c *r.body* f←

r.clause ⟶ *r.guard* [↲ →c *process* f←]

r.guard ⟶ `always` | { `while` *value* }

©2006-10, Dr. Ian Robert East

a.clause ⟶ *a.named.clause* | *a.unnamed.clause*

a.named.clause ⟶ *name* ↲ →c *a.unnamed.clause* f←

a.unnamed.clause ⟶ { *a.guard* ↲ →c *process* f← } | *a.replication*

a.replication ⟶ `for any` *range* ↲ →c *a.unnamed.clause* f←

a.guard ⟶ *communication* | *timing* | `quiet`

‡ *criterion* ⟶ *condition* | *communication*

p.set.item ⟶ *set* ↲ →c *process* f←

p.sn.list ⟶ ↲ →c *p.set.item* { ↲c *p.set.item* }$^+$ f←

set ⟶ *list* | *subrange* | *citation*

condition ⟶ `is` *value*

Semantic rule: IF clauses must be distinct in their guard, though they may share domain.

Semantic rule: Unless a domain shared between clauses is accounted for in its entirety – which may or may not be verified, according to implementation – an otherwise clause is mandatory. Any Boolean domain must be so accounted.

Semantic rule: A condition fails when its result is NULL, but succeeds otherwise.

Semantic rule: The subject of a CASES clause must be scalar and dynamically evaluated.

Semantic rule: No communication may be selective at both ends.

Semantic rule: Every selection must have an opportunity for success, via either communication, finite time delay, or elaboration of an entire domain.

Semantic rule: At least one clause within a repetition must contain a process.

Semantic rule: To simplify translation, each value in any replication range must be static.

Semantic rule: Only one QUIET clause is allowed within nested alternation constructions.

Semantic rule: All communication must accord with an applicable declared interface.

Semantic rule: An object in the context of an alternation may possess at most one keeper.

Semantic rule: To simplify translation, each value in any range must be static.

Implementation note: The internal representation of class must include a record of the size of the applicable domain, or an indication that it is indefinite.

Implementation note: No replication index may ever exceed the given range, which infers a guarantee of remaining within its domain. (This is not the case in some other languages, where an index must *exceed* a stated range before a condition fails and repetition terminates.)

Recursion and reference

recursion ⟶ `process` *name* **:** ↲ *composition*

 ©2006-10, Dr. Ian Robert East

	p.reference	⟶	*name parameters*
‡	*parameters*	⟶	*list* [*;* *list*]

Semantic rule: A recursive composition may be sequential only, never parallel.

Semantic rule: Both naming without recursion and any external reference are disallowed.

Semantic rule: Every external process reference must be to an off-line (item) definition.

Semantic rule: No recursive reference may include configuration parameters.

Semantic rule: Configuration values must be of type NATURAL.

Context

	interface	⟶	[*seq.interface* ↲] [*par.interface* ↲]
	values	⟶	`define` { *equation* | { ↲ →c *equation* { ↲c *equation* }$^+$ f← } } ↲
	state	⟶	`named` { *declaration* | { ↲ →c *declaration* { ↲c *declaration* }$^+$ f← } } ↲
	seq.interface	⟶	[*receipts*] [*loans*] [*returns*]
‡	*equation*	⟶	*name* `value` *name* `as` *value*
	receipts	⟶	`received` { *s.i.declaration* | { ↲ →c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲
	loans	⟶	`borrowed` { *s.i.declaration* | { ↲ →c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲
	returns	⟶	`returned` { *s.i.declaration* | { ↲ →c *s.i.declaration* { ↲c *s.i.declaration* }$^+$ f← } } ↲
	declaration	⟶	*name name*
	s.i.declaration	⟶	*declaration* [*alias*]
	alias	⟶	`alias` *name*

PAR-INTERFACE and NETWORK are defined below.

Semantic rule: A naming may be followed only by a sequence, and a network by a parallel.

Semantic rule: Reference to any object created by a component of a parallel composition must be confined within that component. It must *not* occur in any other.

Semantic rule: Any object borrowed must be assigned. Any value received must be used.

Semantic rule: No ALIAS phrase may appear in the sequential interface of an off-line process definition (item).

HPE requirement: A warning should be issued when a named object is assigned only once.

B.3.4 Network (Honeysuckle Design Language – HDL)

Interface

‡ *par.interface* ⟶ `interface` ↲ →c *port* { ↲c *port* }* f← ↲

‡ *port* ⟶ { { `provider of` } | { `client of` } } *service.id*

service.id ⟶ *name* [`alias` { *name* | `?` }]

Network

‡ *network* ⟶ `network` ↲
→c *n.element* { ↲c *n.element* }$^{+}$ f← ↲

‡ *n.element* ⟶ { *provision* | *dependency* | *selection* | *alternation* }

‡ *provision* ⟶ *name*

dependency ⟶ *provision* > *consumption* { { , | ; } *consumption* }*

selection ⟶ `select` ↲ →c *n.s.element* { ↲c *n.s.element* }$^{+}$ f←

‡ *alternation* ⟶ `alternate` ↲
→c *n.a.element* { ↲c *n.a.element* }$^{+}$ f←

n.s.element ⟶ { *provision* | *dependency* }

‡ *n.a.element* ⟶ { *provision* | *dependency* | *selection* }

‡ *consumption* ⟶ [ǀ] *name*

Design rule: Any additional dependency of the lower-priority service must be arranged in sequence, and not interleaved, with the feedback path when fed forward to further interleaved provision, along with the higher-priority path. The order of the sequence is immaterial.

Design rule: But for services that are sequentially consumed, it must be possible to draw the service digraph so that no two dependencies cross.

Semantic rule: Any dependency may be expressed only once.

©2006-10, Dr. Ian Robert East

Semantic rule: Where a service is consumed internally (entirely within a component), the number of providers must not exceed the number of consumers.

Implementation note: Parsing must include detection of services consumed within the same interstice within the service provided.

B.3.5 Class

‡*class.construction* ⟶ *exposition*

‡ *exposition* ⟶ `is` { *citation* | { ↲ →c *enumeration* f← } }

citation ⟶ { `a` | `an` } *range*

enumeration ⟶ *element* { ↲c `or` *element* }+

‡ *element* ⟶ *constant* | *citation*

constant ⟶ *literal* | *name*

range ⟶ *name* [*subrange*]

subrange ⟶ `from` *value* { { `to` *value* } | { `for` *value* } }

Semantic rule: The choice between `a` and `an` is to be made correctly.

Semantic rule: No object reference is permitted in a class citation subrange.

Semantic rule: A subrange may only be applied to an enumeration, or an equivalence with one.

Semantic rule: The domain of a subrange must be *finite* and lie within a class that is *scalar*.

Semantic rule: The value following TO must share the same class as that following FROM.

Semantic rule: The value following FOR must be of class NATURAL.

Semantic rule: Any class prefixed by EACH must be statically defined (without recursion or dynamically controlled replication).

Semantic rule: Processing of any union must be subject to reduction by selection to basic classes or compositions.

Semantic rule: A universal name-space is maintained against all composition.

Semantic rule: To simplify translation, each value in any range must be static.

Implementation note: Upon importing any process definition, the uniqueness of each element in the universal (and any other) name-space must be verified.

HPE requirement: Programmer must be warned of any overlap between class definitions, as these may be formed by an unintended reuse of a name.

HSE requirement: The following class enumerations are to be predefined:

- NATURAL
- INTEGER
- REAL
- CHARACTER.

Note that each of the above has a corresponding literal lexicon.

B.3.6 Service

‡ *service* ⟶ *s.communication* | *s.sequence*

s.communication ⟶ *s.transfer* | *s.acquire* | *s.send* | *s.receive*

s.transfer ⟶ `transfer` { *object* | *citation* }

s.acquire ⟶ `acquire` { *object* | *citation* }

s.send ⟶ `send` { *value* | *citation* }

s.receive ⟶ `receive` { `signal` | *object* | *citation* }

s.sequence ⟶ `sequence` ↲ →*c* *service* { ↲*c* *service* }$^{+}$ *f*←

B.3.7 Annotation

Annotation employed in HCL is precisely the same as in HPL. Refer to #A.3.7.

©2006-10, Dr. Ian Robert East

Appendix C: Honeysuckle visual design language (HVDL)

C.1 Introduction

It should be apparent that the Honeysuckle Design Language (HDL) – a division of HPL – is capable of describing a system purely in terms of communication. HVDL constitutes a *visual* language, equivalent in that it is capable of expressing precisely the same meaning.

At the level of design, which is capable of independent compilation and verification, there is no need of process abstraction; prioritized service architecture (PSA) is sufficient.

As a result, HVDL omits process 'bubbles'. Where a non-trivial interface exists, *e.g.* a multiple dependency, a vertical line is employed instead.

A PSA module comprises an *abstract interface*, which simply exposes dependencies. Any interface incapable of *reduction* is termed a *physical interface*.

A division of the Honeysuckle Programming Environment (HPE), is called for, which we shall name the *Honeysuckle Design Environment*, or HDE. This will include a *visual editor*, with interactive capability that includes the ability to interrogate a design (*e.g.* recover the textual expression of an interface) and reduce any area of the design to an abstract interface.

C.2 Visual syntax

C.2.1 Independent service

```
network
  s
```

Figure C.1 A single independent service.

The orientation of the arrow is towards the service provider.

Terminals indicate both internal consumption and internal provision.

C.2.2 Sharing and distribution

```
network
  shared console
```

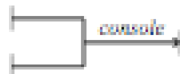

Figure C.2 A service shared by two internal clients.

```
network
  shared distributed validate
```

Figure C.3 A single service both shared and distributed..

C.2.3 Internal interface (dependency)

```
network
  s1 > s2
```

Figure C.4 A single interstitial dependency.

```
network
  s1 > |s2
```

Figure C.5 A single interleaved dependency.

Dependency defines *sequential* structure within a service architecture.

An *interstitial* dependency is completed between two steps of the dependent service.

An *interleaved* dependency sees its steps interleave with those of the dependent service.

```
network
  s1 > s2; s3
```

Figure C.6 Sequential consumption.

```
network
  s1 > s2, s3
```

Figure C.7 Interleaved consumption.

C.2.4 Service selection and alternation

```
network
  select
    s1
    s2 > s4
    s3
```

Figure C.8 Selection between mutually exclusive service provision.

```
network
  interleave
    s1
```


©2006-10, Dr. Ian Robert East

```
s2 > s4
s3
```

Figure C.7 Prioritized alternation of service provision.

C.2.5 External interface and modularity

```
interface
  provider of s
```

Figure C.8 External consumption, internal provision

```
interface
  client of s
```

Figure C.9 External provision, internal consumption

```
interface
  provider of s1
  client of s2
```

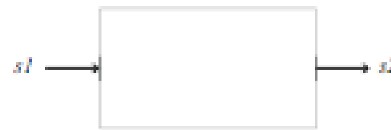

Figure C.10 Visual representation of component interface.

Appendix D: Honeysuckle programming environment (HPE)

D.1 Requirements

The following tools should be available:

–	HVDC	Honeysuckle Visual Design: Capture	capture of graphical design translation to (textual) HDL
–	HDVC	Honeysuckle Design: Verify and Compile	verify design against formal rules compile (translate to binary)
–	HPLC	HPL: Compile	compile (incorporating HDVC).

In addition a text editor (see below) and filing system is required.

The text editor must be capable of reading, editing, and writing, UTF-16-encoded text.

D.2 Recommendations

D.2.1 Folding editor

An element of the Honeysuckle philosophy is that the way in which a program is viewed and constructed should remain orthogonal to both desired behaviour and the way in which it is expressed. A block should be defined as a separate procedure to avoid repetition, not merely so that it can be displayed and considered alone.

Along with occam, came a *folding editor* that permitted a block to collapse, or *fold*, onto a single line. Each fold could be expanded or displayed in isolation. A program could thus be navigated hierarchically and its structure be rendered apparent *at any level*. For example, the outermost algorithm could be rendered transparently even on a display of modest size.

Folding obviates the traditional fragmentation of a program into multiple files for mere manageability. Project modularity need serve only the proper purposes of separated development and reuse.

Many contemporary editors claim to offer folding. However, none attribute independent status to a fold, as a true object in its own right. For example, it is usually impossible to either indent or (convert to) comment a fold. Typically only the first line inside the fold is affected. The true utility of a fold is to effect change to any arbitrary block, of any size, all at once. One should have to write and apply a script to comment out a thousand lines, say.

A proper folding editor is recommended as part of an integrated HPE.

 ©2006-10, Dr. Ian Robert East

Appendix E: Honeysuckle system environment (HSE)

D.1 Requirements

E.1.1 Class (data type) provision

Any HSE should provide definitions of:

- NATURAL 0, 1, 2 … (2^M-1)
- INTEGER -2^{M-1}, … (2^{M-1}-1)
- REAL (*IEEE* standard)
- CHARACTER UTF-16.

M to be that of 16, 32, or 64-bit register, consistently across all numeric classes.

Suitable constraints (invariants) to apply.

E.1.2 Service provision

A minimum of ANSI console service is to be provided.

E.1.3 Library provision

To be defined.

D.2 Recommendations

To be added.

©2006-10, Dr. Ian Robert East

Appendix F: Possible changes and extensions

Recall that this is a *draft* manual. As such it documents a *draft* programming language. The final version of the language will only be composed after a period of digestion, experimentation, and consultation with the *Communicating Process Architectures* (CPA) community.

Two areas of experience are considered *vital* to the development of Honeysuckle:

1. its use as a "publication language" for the composition of as many example programs as possible
2. the progressive development of a compiler.

Any assistance with the above, together with any more immediate observations, would be most welcome and are invited. Please communicate with the author.

This document represents the current state of the language definition and HSE/HPE requirements. Everything herein supercedes anything published previously.

There follows a list of possible language features that are already under consideration:

1. system reset/termination via poison-spreading algorithm of P. H. Welch [10]
2. an ability to render objects persistent
   obviates components which map objects to files
3. machine mapping
   mapping an object or process to a specific memory address
4. an 'oracle' mechanism for the resolution of *n*-way bipolar selection [11].

All of these things could be achieved via additional system structure, at the cost of additional work, at the risk of error, and with a certain loss of transparency.

Automated response to reception of poison is to be adopted. It imposes no obligation or overhead of any kind, save the definition of poison for each data type employed. Which, like NULL can be assigned by default.

Open questions:

1. Whether to allow multiple-exit loops?

   Dijkstra did (in a publication language, and with non-determinism).

   Wirth has considered such for a (private) version of Oberon.

   Standing intention is to allow single-exit only (manual update pending).
2. Whether to allow exemption of one or more consumers from synchronized provision?

   Would add complication to HVDL /HDL.

 ©2006-10, Dr. Ian Robert East

Bibliography

[1] Inmos Ltd.1988. *occam 2 Reference Manual.* Prentice Hall. ISBN 0-13-629312-3.

[2] Maddux, R. 1975. *A Study of Program Structure.* Ph.D. Thesis. University of Waterloo.

[3] Hoare, C. A. R.1975. Recursive data structures. *Int. J. Computer and Information Sciences,* 4(2), pp. 105-132.

[4] Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice Hall. ISBN 0-13-153271-5.

[5] Roscoe, A. W.1998. *The Theory and Practice of Concurrency,* Prentice Hall. ISBN 0-13-674409-5.

[6] Martin, J. M. R. 1996. *The Design and Construction of Deadlock-Free Concurrent Systems.* Ph. D. Thesis, University of Buckingham.

[7] East, I. R. 2004. Prioritized service architecture. Proceedings of Communicating Process Architectures 2004, Series in Concurrent Systems Engineering, pages 55–69. IOS Press, 2004. Isbn 1-58603-458-8.

[8] Unicode Inc. 2003. *The Unicode Standard, Version 4.0.* Unicode Inc. ISBN 0-321-18578-1.

[9] Geoff Barrett. 1992. *occam 3 Reference Manual.* Inmos Ltd.

[10] Welch, P. H. 1989. Graceful termination – graceful resetting. *Proceedings of the 7th Technical Meeting of the occam User Group,* Series in Concurrent Systems Engineering, pages 310-317. ISBN 90-5199-011-1.

[11] Welch, P. H. 2006. A fast resolution of choice between multi-way synchronizations. *Proceedings of Communicating Process Architectures 2006,* Series in Concurrent Systems Engineering, page 389. ISBN 1-58603-671-8.