B Terminology 406

B.1 Term Table of Contents

In any *development project* it is important to define the terms before their first use, to maintain, including adjust, update and extend, such a glossary of term definitions, and to adhere to the definitions.

B.2 Terms

..... $\mathcal{A}$

1. **Abstract:** Something which focuses on essential properties. Abstract is a relation: something is abstract with respect to something else (which possesses — what is considered — inessential properties).

2. **Abstract algebra:** An *abstract*[1] *algebra*[26] is an algebra whose carrier elements and whose functions are defined by *postulate*s (*axiom*[75]s, *law*s) which specify general properties, rather than values, of functions. (Abstract algebras are also referred to as *postulational*, or *axiom*[75]*atic algebras*. The axiomatic approach to the study of algebras forms the cornerstone of so-called modern algebra [159].)

3. **Abstraction:** 'The art of abstracting. The act of separating in thought; a mere idea; something visionary.'

4. **Abstract data type:** An *abstract*[1] *data*[193] *type*[782] is a set of values for which no external world or computer (i.e., data) representation is being defined, together with a set of abstractly defined functions over these data values.

5. **Abstraction function:** An *abstraction*[3] *function*[310] is a function which applies to *value*[802]s of a *concrete type*[157] and yields values of — what is said to be a corresponding — *abstract type*[7]. (Same as *retrieve function*[624].)

6. **Abstract syntax:** An *abstract*[1] *syntax*[733] is a set of rules, often in the form of an *axiom system*[77], or in the form of a set of *sort definition*[695]s, which defines a set of structures without prescribing a precise external world, or a computer (i.e., data) representation of those structures.

7. **Abstract type:** An *abstract*[1] *type*[782] is the same as an *abstract data type*[4], except that no functions over the data values have been specified.

8. **Accessibility:** We say that a *resource*[620] is accessible by another resource, if that other resource can make use of the former resource. (Accessibility is a *dependability requirement*[218]. Usually accessibility is considered a *machine*[436] property. As such, accessibility is (to be) expressed in a *machine requirements*[438] document.)

9. **Acceptor:** An acceptor is a device, like a *finite state automaton*[289] of a *pushdown automaton*[564], which, when given (i.e., presented with) character strings (or, in general, finite structures), purported to belong to a language, can recognise, i.e., can decide, whether these character strings belong to that language.

10. **Acquirer:** The legal entity, a person, an institution or a firm which orders some *development*[228] to take place. (Synonymous terms are *client*[116] and *customer*[192].)

11. **Acquisition:** The common term means purchase. Here we mean the collection of *knowledge*[407] (about a *domain*[239], about some *requirements*[605], or about some *software*[685]). This collection takes place in an interaction between the *developer*[227]s and representatives of the *client*[116] (*user*[796]s, etc.). (A synonym term is *elicitation*[265].)

12. **Action:** By an action we shall understand something which potentially changes a *state*[705], that is, *value*[802]s of *dynamic*[260] *attribute*[69]s of *simple entities*[681]. We consider *action*[12]s to be one of the four kinds of *entities*[272] that the `Triptych` "repeatedly" considers. The other three are: *simple entities*[681], *event*[281]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.

13. **Activation stack:** See the *Comment* field of the *function activation*[311] entry.

14. **Active:** By active is understood a *phenomenon*[524] which, over *time*[761], changes *value*[802], and does so either by itself, *autonomous*[73]ly, or also because it is "instructed" (i.e., is "bid" (see *biddable*[85]), or "programmed" (see *programmable*[546]) to do so). (Contrast to *inert*[367] and *reactive*[578].)

15. **Actor:** By an actor we shall understand someone which carries out an *action*[12]. (A synonymous term for actor is *agent*[24].)

16. **Actual argument:** When a function is invoked it is usually applied to a list of values, the actual *argument*[52]s. (See also *formal parameter*[302].)

17. **Actuator:** By an actuator we shall understand an electronic, a mechanical, or an electromechanical device which carries out an *action*[12] that influences some physical *value*[802]. (Usually actuators, together with *sensor*[659]s, are placed in *reactive*[578] systems, and are linked to *controller*[183]s. Cf. *sensor*[659].)

18. **Acyclic:** Acyclicity is normally thought of as a property of graphs. (Hence see next entry: *acyclic graph*[19].)

19. **Acyclic graph:** An acyclic graph is usually thought of as a *directed graph*[232] in which there is no nonempty *path*[517], in the direction of the *arrow*[54]s, from any *node*[479] to itself. (Often acyclic graphs are called directed acyclic graphs, *DAG*s. An undirected graph which is acyclic is a *tree*[777].)

20. **Adaptive:** By adaptive we mean some thing that can adapt or arrange itself to a changing *context*[172], a changing *environment*[275].

21. **Adaptive maintenance:** By adaptive maintenance we mean an update, as here, of software, to fit (to adapt) to a changing environment. (Adaptive maintenance is required when new input/output media are attached to the existing software, or when a new, underlying database management system is to be used (instead of an older such), etc. We also refer to *corrective maintenance*[187], *perfective maintenance*[519], and *preventive maintenance*[541].)

22. **Address:** An address is the same as a *link*[425], a *pointer*[528] or a *reference*[587]: Something which refers to, i.e., designates something (typically something else). (By an address we shall here, in a narrow sense, understand the *location*[431], the place, or position in some *storage*[715] at which some *data*[193] is *store*[714]d or kept.)

23. **Ad hoc polymorphism:** See Comment field of *polymorphic*[529].

24. **Agent:** By an agent we mean the same as an *actor*[15] — a human or a machine (i.e., robot). (The two terms *actor*[15] and *agent*[24] are here considered to be synonymous.)

25. **AI:** Abbreviation for artificial intelligence. (We shall refrain from positing (including risking) a definition of the term AI. Instead we refer to John McCarthy's home page [169].)

26. **Algebra:** An algebra is here taken to just mean: A set of *value*[802]s, A, the *carrier* of the algebra, and a set of *function*[310]s, Φ, on these values such that the result values are within the set of values: $\Phi = \mathsf{A}^* \rightarrow \mathsf{A}$. (We make the distinction between *universal algebra*[790]s, *abstract algebra*[2]s and *concrete algebra*[155]s. See also *heterogeneous algebra*[336]s, *partial algebra*[515]s and *total algebra*[765]s.)

27. **Algebraic semantics:** By an algebraic semantics we understand a *semantics*[655] which denotes one, or a (finite or infinite) set of zero, one or more *algebra*[26]s. (Usually an algebraic semantics is expressed in terms of (i) *sort*[694] definitions, (ii) *function signature*[318]s and (iii) *axiom*[75]s.)

28. **Algebraic systems:** An algebraic system is an *algebra*[26]. (We use the term *system*[736] as an entity with two clearly separable parts: the *carrier*[106] of the algebra and the *function*[310]s of the algebra. We distinguish between *concrete algebra*[155]s, *abstract algebra*[2]s and *universal algebra*[790]s — here listed in order of increasing *abstraction*[3].)

29. **Algebraic type:** An algebraic type is here considered the same as a *sort*[694]. (That is, algebraic types are specified as are *algebraic systems*[28].)

30. **Algol:** Algol stands for Algorithmic Language. (`Algol 60` designed in the period 1958–1960 [12]. It became a reference standard for future language designs (Algol W [233], Algol 68 [224], Pascal [230, 129, 141] and others.)

31. **Algorithm:** The notion of an algorithm is so important that we will give a number of not necessarily complementary definitions, and will then discuss these.

    - By an algorithm we shall understand a precise prescription for carrying out an orderly, finite set of *operation*[493]s on a set of *data*[193] in order to calculate (*compute*[148]) a result. (This is a version of the classical definition. It is compatible with computability in the sense of *Turing machine*[781]s and *Lambda-calculus*[412]. Other terms for algorithm are: effective procedure, and abstract program.)
    - Let there be given a possibly infinite set of *state*[705]s, S, let there be given a possibly infinite set of initial states, I, where $I \subseteq S$, and let there be given a next state function $f : S \to S$. (C, where $C = (Q, I, f)$ is an initialised, *deterministic*[226] *transition*[772] system.) A sequence $s_0, s_1, \ldots, s_{i-1}, s_i, \ldots, s_m$ such that $f(s_{i-1}) = s_i$ is a *computation*[144]. An algorithm, A, is a C with final states O, i.e.: $A = (Q, I, f, O)$, where $O \subseteq S$, such that each computation ends with a state s_m in O. (This is basically Don Knuth's definition [144]. In that definition a state is a collection of identified data, i.e., a formalised representation of information, i.e., of computable data. Thus Knuth's definition is still Turing and Lambda-calculus "compatible".)
    - There is given the same definition as just above with the generalisation that a state is any association of variables to phenomena, whether the latter are representable "inside" the computer or not. (This is basically Yuri Gurevitch's definition of an algorithm [117, 198, 199]. As such this definition goes beyond Turing machine and Lambda-calculus "compatibility". That is, captures more!)

32. **Algorithmic:** Adjective form of *algorithm*[31].

33. **Allocate:** To apportion for a specific purpose or to particular persons or things, to distribute tasks among human and automated components. (We shall here use the term generally for the allocation of *resources* (see also *resource allocation*[621]), specifically for *storage*[715] to *assignable variable*[59]s. In the general sense, allocation, as the name implies, has some spatial qualities about it: allocation to spatial positions. In the special sense we can indeed talk of storage space.)

34. **Alphabet:** A finite collection of script symbols called the letters of the alphabet.

35. **Alpha-renaming:** By alpha-renaming (α-renaming) we mean the substitution of a *binding*[88] *identifier*[351], with another, the "new", identifier, in some *Lambda-expression*[414]

(statement or clause), such that all free occurrences of that binding identifier in that expression (statement or clause) are replaced by the new identifier, and such that that new identifier is not already bound in that expression (statement or clause). (Alpha-renaming is a concept of the *Lambda-calculus*[412].)

36. **Ambiguous:** A *sentence*[660] is ambiguous if it is open to more than one *interpretation*[397], i.e., has more than one *model*[460] and these models are not *isomorphic*[403].

37. **Analogic:** Equivalency or likeness of relations. Resemblance of relations or attributes as a ground of reasoning. Also: Presumptive reasoning based on the assumption that if things have some similar attributes, their other attributes will be similar [160].

38. **Analogue:** A representative in another class or group [160]. (Used in this technical note in the sense above, not in the sense of electrical engineering or control theory.)

39. **Analysis:** The resolution of anything complex into simple elements. A determination of proper components. The tracing of things to their sources; the discovery of general principles underlying concrete phenomena [160]. (In conventional mathematics analysis pertains to continuous phenomena, e.g. differential and integral calculi. Our analysis is more related to hybrid systems of both discrete and continuous phenomena, or often to just discrete ones.)

40. **Analytic:** Of, or pertaining to, or in accordance with *analysis*[39].

41. **Analytic grammar:** A *grammar*[325], i.e., a *syntax*[733] whose designated sentences (in general: Structures) can be subject to *analysis*[39], i.e., where the syntactic composition can be revealed through *analysis*[39].

42. **Anomaly:** Deviation from the normal.

43. **Anthropomorphic:** Attributing a human personality to anything impersonal or irrational [160]. (See *anthropomorphism*[44]. It seems to be a "disease" of programmers to attribute their programs with human properties: "The program does so-and-so; and after that, it then goes on to do such-and-such," etcetera. Programs, to recall, are, as are any description is, a mere syntactic, i.e., static text. As such they certainly can "do nothing". But they may prescribe that certain actions are effected by machine — when a machine interprets ("executes") the program text!)

44. **Anthropomorphism:** Ascription of a human form and attributes to the Deity, or of a human attribute or personality to anything impersonal or irrational [160]. (See *anthropomorphic*[43].)

45. **Application:** By an application we shall understand either of two rather different things: (i) the application of a function to an *argument*[52], and (ii) the use of software for some specific purpose (i.e., the application). (See next entry for variant (ii).)

46. **Application domain:** An area of activity which some *software*[685] is to support (or supports) or partially or fully automate (resp. automates). (We normally omit the prefix 'application' and just use the term *domain*[239].)

47. **Applicative:** The term applicative is used in connection with applicative programming. It is hence understood as programming where applying functions to *argument*[52]s is a main form of expression, and hence designates function application as a main form of operation. (Thus the terms applicative and *functional*[312] are here used synonymously.)

48. **Applicative programming:** See the term *applicative*[47] just above. (Thus the terms applicative programming and *functional programming*[313] are here used synonymously.)

49. **Applicative programming language:** Same as *functional programming language*[314].

50. **Arc:** Same as an *edge*[262]. (Used normally in connection with *graph*[327]s.)

51. **Architecture:** The structure and content of *software*[685] as perceived by their *user*[796]s and in the context of the *application domain*[46]. (The term architecture is here used in a rather narrow sense when compared with the more common use in civil engineering.)

52. **Argument:** A *value*[802] provided (possibly as part of an argument list) when invoking a function.

53. **Arity:** By the arity of a *function*[310] (i.e., an *operation*[493]) we understand the number (0, 1, or more) of *argument*[52]s that the function applies to. (Usually a function applies to an argument list, and the arity is therefore the length of this list.)

54. **Arrow:** A directed *edge*[262]. (*Branches* are arrows.)

55. **Artefact:** An artificial product [160]. (Anything designed or constructed by humans or machines, which is made by humans.)

56. **Artifact:** Same term as *artefact*[55].

57. **Artificial intelligence:** See *AI*[25].

58. **Assertion:** By an assertion we mean the act of stating positively usually in anticipation of denial or objection. (In the context of *specification*[698]s and *program*[545]s an assertion is usually in the form of a pair of *predicate*[536]s "attached" to the specification text, to the program text, and expressing properties that are believed to hold before any interpretation of the text; that is, "a before" and "an after", or, as we shall also call it: a **pre-** and a **post-**condition.)

59. **Assignable variable:** By an assignable variable we understand an entity of a program text which *denote*[216]s a *storage*[715] *location*[431] whose associated *value*[802] can be changed by an *assignment*[60]. (Usually, in the context of specifications and programs, assignable variables are declared.)

60. **Assignment:** By an assignment we mean an update to, a change of a *storage*[715] *location*[431]. (Usually, in the context of specifications and programs, assignments are prescribed by assignment statements.)

61. **Associative:** Property of a binary operator o: If for all values a, b and c, $(a\ o\ b)\ o\ c = a\ o\ (b\ o\ c)$, then o is said to be an associative operator. (Addition (+) and multiplication (*) of natural numbers are associative operators.)

62. **Asynchronous:** Not *synchronous*[731]. (In the context of computing we say that two or more *process*[544]es — some of which may represent the world external to the computing device — are asynchronous if occurrences of the *event*[281]s of these processes are not (a priori) coordinated.)

63. **Atomic:** In the context of *software engineering*[693] atomic means: A *phenomenon*[524] (a *concept*[152], a *simple entity*[681], a *value*[802]) which consists of no proper subparts, i.e., no proper sub*phenomena*[524], sub*concept*[152]s, sub*entities*[272] or sub*value*[802]s other than itself. When we consider a *phenomenon*[524], a *concept*[152], a *simple entity*[681], a *value*[802], to be atomic, then it is often a matter of choice, with the choice reflecting a level of *abstraction*[3].

64. **Atomic action:**

65. **Atomic behaviour:**

66. **Atomic entity:** Either an *atomic action*[64], an *atomic behaviour*[65], an *atomic event*[67] or an *atomic simple entity*[68]

67. **Atomic event:**

68. **Atomic simple entity:**

69. **Attribute:** We use the term attribute only in connection with values of composite type. An attribute is now whether a composite value possesses a certain property, or what value it has for a certain component part. (An example is that of database (e.g., SQL) relations (i.e., tabular data structures): Columns of a table (i.e., a relation) are usually labelled with a name designating the attribute (type) for values of that column. Another example is that, say, of a Cartesian: A = B×C×D. A can be said to have the attributes B, C, and D. Yet other examples are M = A $\xrightarrow[m]{}$ B, S = A-**set** and L = A*. M is said to have attributes A and B. S is said to have attribute A. L is said to have attribute A. In general we make the distinction between an entity consisting of subentities (being decomposable into proper parts, cf. *subentity*[721]), and the entities

having attributes. A person, like me, has a height attribute, but my height cannot be "composed away from me"!)

70. **Attribute grammar:** A grammar, usually expressed as a *BNF Grammar*[92], where, to each *rule*[638], and to each nonterminal, of the left-hand side or of the right-hand side of the rule, there is associated one or more (attribute) *assignable variable*[59]s together with a set of single assignments to some of these variables — such that the assignment expression variables are those of the attribute variables of the rule.

71. **Automaton:** An automaton is a device with *state*[705]s, *input*[382]s, some states designated as final states, and with a next state *transition*[772] function which to every state and input designates a next state. (There may be a finite, or there may be an infinite number of states. The next state transition function may be *deterministic*[226] or *nondeterministic*[481].)

72. **Automorphism:** An *isomorphism*[404] that maps an algebra into itself is an automorphism. (See also *endomorphism*[268], *epimorphism*[276], *homomorphism*[343], *monomorphism*[467].)

73. **Autonomous:** A *phenomenon*[524] (a *concept*[152], an *entity*[272]) is said to be autonomous if it changes *value*[802] at its own discretion or without influence from an *environment*[275]. (Rephrasing the above we get: (i) A phenomenon is said to be of, or possess, the autonomous active dynamic attribute if it changes value only on its own volition — that is, it cannot also change value as a result of external stimuli; (ii) or when its actions cannot be controlled in any way: That is, they are a "law onto themselves and their surroundings". We speak of such *phenomena* as being *dynamic*[260]. Other dynamic *active*[14] phenomena may be *active*[14] or *reactive*[578].)

74. **Availability:** We say that a *resource*[620] is available for use by other resources, if within a reasonable time interval these other resources can make use of the former resource. (Availability is a *dependability requirement*[218]. Usually availability is considered a *machine*[436] property. As such availability is (to be) expressed in a *machine requirements*[438] document.)

75. **Axiom:** An established rule or principle or a self-evident truth.

76. **Axiomatic specification:** A *specification*[698] presented, i.e., given, in terms of a set of *axiom*[75]s. (Usually an axiomatic specification also includes definitions of *sort*[694]s and *function signature*[318]s.)

77. **Axiom system:** Same as *axiomatic specification*[76].

$\mathcal{B}$

78. **B:** B stands for Bourbaki, pseudonym for a group of mostly French mathematicians which began meeting in the 1930s, aiming to write a thorough unified set-theoretic

account of all mathematics. They had tremendous influence on the way mathematics has been done since. (The founding of the Bourbaki group is described in André Weil's autobiography, titled something like "memoir of an apprenticeship" (orig. Souvenirs D'apprentissage). There is a usable book on Bourbaki by J. Fang. Liliane Beaulieu has a book forthcoming, which you can sample in "A Parisian Cafe and Ten Proto-Bourbaki Meetings 1934–1935" in the Mathematical Intelligencer 15 no. 1 (1993) 27–35. From `http://www.faqs.org/faqs/sci-math-faq/bourbaki/` (2004). Founding members were: Henri Cartan, Claude Chevalley, Jean Coulomb, Jean Delsarte, Jean Dieudonné, Charles Ehresmann, René de Possel, Szolem Mandelbrojt, André Weil. From: `http://www.bourbaki.ens.fr/` (2004). `B` also stands for a model-oriented specification language [2].)

79. **Behaviour:** A sequence of *action*[12]s and *event*[281]s is a behaviour. A set of behaviours is a behaviour.

    By behaviour we shall understand the way in which something functions or operates.

    In the context of domain engineering behaviour is a concept associated with *phenomena*[524], in particular manifest *simple entities*[681]. And then behaviour is that which can be observed about the *value*[802] of that *simple entity*[681] and its *interaction*[392] with its *environment*[275].

80. **Behaviour, Communicating:** A concurrent behaviour where actions of one behaviour synchronise and communicate with actions of other behaviours.

81. **Behaviour, Concurrent:** A set of behaviours.

82. **Behaviour, Parallel:** A set of behaviours.

83. **Behaviour, Sequential:** A sequence of actions and events.

84. **Beta-reduction:** By Beta-reduction we understand the substitution whereby all *free*[305] occurrences of a designated *variable*[803] in a *Lambda-expression*[414] are replaced by *Lambda-expression*[414] (in which some *Alpha-renamings* may have to be made first).

85. **Biddable:** A *phenomenon*[524] is biddable if it can be advised (through a "contractual arrangement") on which *action*[12]s are expected of it in various *state*[705]s. (A biddable phenomenon does not have to take these actions, but then the "contractual arrangement" need no longer be honoured by other phenomena (other [sub]domains) with which it *interact*[391]s (i.e., shares phenomena).)

86. **Bijection:** See *bijective function*[87].

87. **Bijective function:** A total *surjective function*[727] which maps all *value*[802]s of its postulated *definition set*[211] into all distinct values of its postulated *range*[576]set is called bijective. (See also *injective function*[380] and *surjective function*[727].)

88. **Binding:** By binding we mean a pairing of, usually, an *identifier*[351], a *name*[474], with some *resource*[620]. (In the context of software engineering we find such bindings as: (i) of an *assignable variable*[59] to a *storage*[715] *location*[431], (ii) of a *procedure*[543] *name*[474] to a procedure *denotation*[213], etc.)

89. **Block:** By a block we shall here understand a textual entity, one that is suitably delineated. (In the context of software engineering a block is normally some partial *specification*[698] which locally introduces some (*applicative*[47], i.e., expression) constant definitions (i.e., **let .. in .. end**), or some (*imperative*[352], i.e., statement) local variable declarations (i.e., **begin dcl .. ; .. end**).)

90. **Block-structured programming language:** A *programming language*[551] is said to be block-structured if it permits such program constructs (incl. *procedures*) whose *semantics*[655] amount to the creation of a local identifier *scope*[649], and where such can be nested, zero, one or more within another.

91. **BNF:** Abbreviation for Backus–Naur Form (Grammar). (See *BNF Grammar*[92].)

92. **BNF Grammar:** By BNF Grammar we mean a concrete, linear textual representation of a *grammar*[325], i.e., a *syntax*[733], one that *designate*[222]s a set of strings. (A BNF Grammar usually is represented in the form of a set of *rule*[638]s. Each rule has a *nonterminal*[484] left-hand-side *symbol*[728] and a finite set of zero, one or more alternative right-hand-side strings of *terminal*[750] and nonterminal symbols.)

93. **Boolean:** By Boolean we mean a data type of logical values (**true** and **false**), and a set of connectives: $\sim$, $\wedge$, $\vee$, and $\Rightarrow$. (Boolean derives from the name of the mathematician George Boole.)

94. **Boolean connective:** By a *Boolean*[93] *connective*[167] we mean either of the Boolean operators: $\wedge$, $\vee$, $\Rightarrow$ (or $\supset$), $\sim$ (or $\neg$).

95. **Bound:** The concept of being bound is associated with (i) *identifier*[351]s (i.e., *name*[474]s) and *expression*[282]s, and (ii) with *name*[474]s (i.e., *identifier*[351]s) and *resource*[620]s. An identifier is said to be either *free*[305] or bound in an expression based on certain rules being satisfied or not. If an identifier is bound in an expression then bound occurrences of that identifier are bound to the same resource. If a name is bound to some resource then all bound occurrences of that name *denote*[216] that resource. (Cf. *free*[305].)

96. **BPR:** See *business process reengineering*[101]

97. **Branch:** Almost the same as an *edge*[262], except that branches are directed, i.e., are (like) *arrow*[54]s. (Used usually in connection with *tree*[777]s.)

98. **Brief:** By a brief is understood a *document*[237], or a part of a document which informs about a *phase*[523] , or a *stage*[702] , or a *step*[711] of *development*[228]. (A brief thus contains *information*[373].)

99. **Business process:** By a business process we shall understand a *behaviour*[79] of an enterprise, a business, an institution, a factory. (Thus a business process reflects the ways in which a business conducts its affairs, and is a *facet*[285] of the *domain*[239]. Other facets of an enterprise are those of its *intrinsics*[399], *management and organisation*[445] (a facet closely related, of course, to business processes), *support technology*[725], *rules and regulations*[640], and *human behaviour*[345].)

100. **Business process engineering:** By *business process engineering*[100] we shall understand the *design*[221], the determination, of *business process*[99]es. (In doing business process engineering one is basically designing, i.e., prescribing entirely new business processes.)

101. **Business process reengineering:** By *business process reengineering*[101] we shall understand the re*design*[221], the change, of *business process*[99]es. (In doing business process reengineering one is basically carrying out *change management*[109].)

. . . . . $\mathcal{C}$

102. **Calculate:** Given an expression and an applicable *rule*[638] of a *calculus*[104], to change the former expression into a resulting expression. (Same as *compute*[148].)

103. **Calculation:** A sequence of steps which, from an initial expression, following rules of a *calculus*[104], *calculate*[102]s another, perhaps the same, expression. (Same as *computation*[144].)

104. **Calculus:** A method of *computation*[144] or *calculation*[103] in a special notation. (From mathematics we know the differential and the integral calculi, and also the Laplace calculus. From metamathematics we have learned of the λ-calculus. From logic we know of the Boolean (propositional) calculus.)

105. **Capture:** The term capture is used in connection with *domain knowledge*[254] (i.e., *domain capture*[242]) and with *requirements acquisition*[606]. It shall indicate the act of acquiring, of obtaining, of writing down, domain knowledge, respectively requirements.

106. **Carrier:** By a carrier is understood a, or the set of *entities* of an *algebra*[26] — the former in the case of a *heterogeneous algebra*[336].

107. **Cartesian:** By a Cartesian is understood an ordered product, a fixed grouping, a fixed composition, of *entities*. (Cartesian derives from the name of the French mathematician René Descartes.)

108. **C.C.I.T.T:** Abbreviation for Comité Consultative Internationale de Telegraphie et Telephonie. (CCITT is an alternative form of reference.)

109. **Change management:** Same as *business process reengineering*[101].

110. **Channel:** By a channel is understood a means of *interaction*[392], i.e., of *communication*[122] and possibly of *synchronisation*[729] between *behaviour*[79]s. (In the context of computing we can think of channels as being either input, or output, or both input and output channels.)

111. **Chaos:** By **chaos** we understand the totally undefined *behaviour*[79]: Anything may happen! (In the context of computing **chaos** may, for example, be the *designation*[223] for the never-ending, the never-terminating *process*[544].)

112. **CHI:** Abbreviation for Computer Human Interface. (Same as *HCI*[334].)

113. **CHILL:** Abbreviation for CCITT's High Level Language. (See [62, 118].)

114. **Class:** By a class we mean either of two things: a **class** *clause*[115], as in RSL, or a set of *entities* defined by some *specification*[698], typically a *predicate*[536].

115. **Clause:** By a clause is meant an *expression*[282], designating a *value*[802], or a *statement*[707], designating a *state*[705] change, or a sentential form, which designates both a value and a state change. (When we use the term clause we mean it mostly in the latter sense of both designating a value and a side effect.)

116. **Client:** By a client we mean any of three things: (i) The legal body (a person or a company) which orders the development of some software, or (ii) a *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., the *server*[663]), in order to have that server perform some *action*[12]s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the first or in the second sense (i, ii).)

117. **Closure:** By a closure is usually meant some transitive closure of a relation $\Re$: If $a\Re b$ and $b\Re c$ then $a\Re c$, and so forth. To this we shall add another meaning, used in connection with implementation of (for example) procedures: Denotationally a procedure, when invoked, in some calling environment, is to be interpreted in the defining environment. Hence a procedure closure is a pair: The procedure text and the defining environment.

118. **Code:** By code we mean a *program*[545] which is expressed in the machine language of a computer.

119. **Coding:** By coding we shall here, simply, mean the act of programming in a machine, i.e., in a computer-close language. (Thus we do not, except where explicitly so mentioned, mean the encoding of one string of characters into another, say for *communication*[122] over a possibly faulty communication *channel*[110] (usually with the decoding of the encoded string "back" into the original, or a similar string).)

120. **Cohesion:** Cohesion expresses a measure of "closeness", of "dependency", of "sticking together" among a set of entities. (In the context of software engineering cohesion

is, as it is here, a term used to express a dependency relation between *module*[464]s of a *specification*[698] or a *program*[545]. Two modules have a higher cohesion the larger the number of cross-references (to types and values, including, in particular functions) there are among them.)

121. **Collision:** Collision, as used here, means that two (or more) occurrences of the same identifier, of which at least one is free, and which at some stage occurred in different text parts, are brought together, say by function application (i.e., macro-expansion) and thereby become bound. (Collision is a concept introduced in the Lambda-calculus, see . Collision is an undesirable effect. See also *confusion*[162].)

122. **Communication:** A *process*[544] by which *information*[373] is exchanged between individuals (*behaviour*[79]s, *process*[544]es) through a common *system*[736] of *symbol*[728]s, *sign*[679]s, or *protocol*[561]s.

123. **Commutative:** Property of a binary operator o: If for all values a and b, $a\ o\ b = b\ o\ a$, then o is said to be a commutative operator. (Addition (+) and multiplication (*) of natural numbers are commutative operators.)

124. **Compilation:** By a compilation we shall mean the conversion, the *translation*[775], of one formal text to another, usually a high-level program text to a low-level machine code text.

125. **Compiler:** By a compiler we understand a device (usually a software package) which given *sentence*[660]s (i.e., *source program*[696]s) in one language, generates sentences (i.e., *target program*[743]s) in another language. (Usually the source and the target languages are related as follows: The source language is normally a so-called "higher-order" language, like Java, and the target language is normally a "lower (abstraction) level" language, like Java Byte Code (or a computer machine language) for which an interpreter is readily available.)

126. **Compiler dictionary:** By a compiler dictionary we shall understand a composite data structure (with a varying number of entries) and a fixed number of operations. The data structure values reflect properties of a program text being compiled. These properties could be: types of some program text variable, type structure of some program text type name, program point of definition of some (goto) label, etc. The possibly hierarchical, i.e., recursively nested, structure of the compiler dictionary further reflects a similarly hierarchical structure of the program text being compiled. The operations include those that insert, update, and search for entries in the compiler dictionary.

127. **Compile time:** By compile time we understand that time interval during which a *source program*[696] is being compiled and during which certain analyses, and hence decisions, can be made about, and actions taken with respect to the source program

(to be, i.e., being, compiled) — such as *type check*[783]ing, name *scope check*[650]ing, etc. (Contrast to *run time*[641].)

128. **Compiling algorithm:** By a compiling algorithm we shall understand a specification which, for every rule in a syntax (of a *source program*[696]ming language), prescribes which *target program*[743]ming language data structure to generate. (We refer to (Sects. 16.8–16.10) for "our story" on compiling algorithms.)

129. **Complete:** We say that a *proof system*[557] is complete if all true sentences are provable.

130. **Completeness:** Noun form of the *complete*[129] adjective.

131. **Component:** By a component we shall here understand a set of type definitions and component local variable declarations, i.e., a component local state, this together with a (usually complete) set of modules, such that these modules together implement a set of concepts and facilities, i.e., functions, that are judged to relate to one another.

132. **Component design:** By a component design we shall understand the *design*[221] of (one or more) *component*[131]s. (We shall refer to for "our story" on component design.)

133. **Composite:** We say that a *phenomenon*[524] or a *concept*[152], is composite when it is possible and meaningful to consider that phenomenon or concept as analysable into two or more subphenomena or subconcepts.

134. **Composite action:**

135. **Composite behaviour:**

136. **Composite entity:** Either a *composite action*[134], a *composite behaviour*[135], a *composite event*[137] or a *composite simple entity*[138].

137. **Composite event:**

138. **Composite simple entity:**

139. **Composite type:**

140. **Composition:** By composition we mean the way in which a *phenomenon*[524], a *concept*[152], is "put together" (i.e., composed) into a *composite*[133] *phenomenon*[524], resp. *concept*[152].

141. **Compositional:** We say that two or more *phenomena* or *concepts* are compositional if it is meaningful to *compose* these phenomena and/or concepts. (Typically a *denotational semantics*[215] is expressed compositionally: By composing the semantics of sentence parts into the semantics of the composition of the sentence parts.)

142. **Compositional documentation:** By compositional documentation we mean a development, or a presentation (of that development), of, as here, some *description*[220] (*prescription*[540] or *specification*[698]), in which some notion of "smallest", i.e., atomic phenomena and concepts are developed (resp. presented) first, then their compositions, etc., until some notion of full, complete development (etc.) has been achieved. (See also *composition*[140], *compositional*[141] and *hierarchical documentation*[340].)

143. **Comprehension:** By comprehension we shall here mean *set*[664], *list*[428] or *map*[449] comprehension, that is, the expression, of a set, a list, respectively a map, by a predicate over the elements of the set, list or pairings of the map, that belong to the set, list, respectively the map.

144. **Computation:** See *calculation*[103].

145. **Computational linguistics:** The study and knowledge of the *syntax*[733] and *semantics*[655] of *language*[417] based on notions of *computer science*[149] and *computing science*[150]. (Thus computational linguistics emphasises those aspects of language whose analysis (*recognition*), or synthesis (*generation*), can be mechanised.)

146. **Computational data+control requirements:** By a computational data + control requirements we mean a requirements which express how the dynamics of computations or data (may) warrant interaction between the machine and its environment, hence is an *interface requirements*[394] *facet*[285]. (See also *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

147. **Computational semantics:** By a computational semantics we mean a specification of the semantics of a language which emphasises run-time computations, i.e., state-to-next-state transitions, as effected when following the prescriptions of programs. (Terms similar in meaning to computational semantics are *operational semantics*[496] and *structural operational semantics*[720].)

148. **Compute:** Given an expression and an applicable *rule*[638] of a *calculus*[104], to change the former expression into a resulting expression. (Same as *calculate*[102].)

149. **Computer Science:** The study and knowledge of the phenomena that can exist inside computers.

150. **Computing Science:** The study and knowledge of how to construct those phenomena that can exist inside computers.

151. **Computing system:** A combination of *hardware*[331] and *software*[685] that together make meaningful *computation*[144]s possible.

152. **Concept:** An abstract or generic idea generalised from phenomena or concepts. (A working definition of a concept has it comprising two components: The *extension*[283] and the *intension*[389]. A word of warning: Whenever we describe something claimed to be a "real instance", i.e., a physical *phenomenon*[524], then even the description becomes that of a concept, not of "that real thing"!)

153. **Concept formation:** The forming, the enunciation, the *analysis*[39], and definition of *concepts* (on the basis, as here, of *analysis*[39] of the *universe of discourse*[793] (be it a *domain*[239] or some *requirements*[605])). (Domain and requirements concept formation(s) is treated in Vol. 3, Chaps. 13 (Domain Analysis and Concept Formation) and 21 (Requirements Analysis and Concept Formation).)

154. **Concrete:** By concrete we understand a *phenomenon*[524] or, even, a *concept*[152], whose explication, as far as is possible, considers all that can be observed about the phenomenon, respectively the concept. (We shall, however, use the term concrete more loosely: To characterise that something, being specified, is "more concrete" (possessing more properties) than something else, which has been specified, and which is thus considered "more abstract" (possessing fewer properties [considered more relevant]).)

155. **Concrete algebra:** A *concrete*[154] *algebra*[26] is an algebra whose carrier is some known set of mathematical elements and whose functions are known, i.e., well-defined. That is, the *model*[460]s of both the carrier and all the functions are pre-established. (Concrete algebras are the level of the empirical (actual) world of mathematics and its applications, where one deals with specific sets of elements (integers, Booleans, reals, etc.), and where operations on these sets that are defined by rules or algorithms or combinations. In general one "knows" a concrete algebra when one knows what the elements of the carrier A are and how to *evaluate*[279] the functions $\phi_i : \Phi$ over A [159].)

156. **Concrete syntax:** A *concrete*[154] *syntax*[733] is a syntax which prescribes actual, computer representable *data structure*[199]s. (Typically a *BNF Grammar*[92] is a concrete syntax.)

157. **Concrete type:** A *concrete*[154] *type*[782] is a type which prescribes actual, computer representable *data*[193] *structure*[719]s. (Typically the type definitions of programming languages designate concrete types.)

158. **Concurrency:** By concurrency we mean the simultaneous existence of two or more *behaviour*[79]s, i.e., two or more *process*[544]es. (That is, a *phenomenon*[524] is said to exhibit concurrency when one can analyse the phenomenon into two or more *concurrent*[159] phenomena.)

159. **Concurrent:** Two (or more) *event*[281]s can be said to occur concurrently, i.e., be concurrent, when one cannot meaningfully describe any one of these events to ("always")

"occur" before any other of these events. (Thus concurrent systems are systems of two or more processes (behaviours) where the simultaneous happening of "things" (i.e., events) is deemed beneficial, or useful, or, at least, to take place!)

160. **Configuration:** By a configuration we shall here understand the *composition*[140] of two or more *semantic value*[802]s. (Usually we shall decompose a configuration into parts such that each part enjoys a *temporal*[747] relationship with respect to the other parts: being "more *dynamic*[260]", being "more *static*[708]", etc. More specifically, we shall typically model the semantics of *imperative*[352] programming languages in terms of *semantic function*[656]s over configurations composed from *environment*[275]s and *storage*[715]s.)

161. **Conformance:** Conformance is a relation between two *document*[237]s (A and B). B is said to conform to A, if everything A specifies is satisfied by B. (Conformance is thus, here, taken to be the same as *correct*[185]ness, i.e., *congruence*[163]. Usually conformance is used in standardisation documents: *Any system claiming to follow this standard must show conformance to it.*)

162. **Confusion:** Confusion, as used here, means that two (or more) occurrences of the same identifier, bound to possibly different values, may be confused in that it is difficult from a smaller context of the text in which they occur to discern, to decide, which meanings, which values, the various occurrences are bound to. (Confusion is a concept introduced in the Lambda-calculus, see . Confusion is an OK, albeit annoying, effect! See also *collision*[121].)

163. **Congruence:** An *algebra*[26], A, is said to be congruent with another algebra, B, if, for every operation, o_B, and suitable set of arguments, b_1, b_2, ..., b_n, to that operation, in B, there corresponds an operation, o_A, and a suitable set of arguments, a_1, a_2, ..., a_n, in A such that $o_A(a_1, a_2, \ldots, a_n) = o_B(b_1, b_2, \ldots, b_n)$. (Compare this definition to that of *conformance*[161]. The difference is one between a precise, mathematical meaning of congruence, as contrasted to an informal meaning of conformance.)

164. **Conjunction:** Being combined, being conjoined, composed. (We shall mostly think of conjunction as the (meaning of the) logical connective "and": $\wedge$.)

165. **Connection:** Connection is a topological notion, and, as such, is also an ontological concept related to "parts and wholes", where parts may be, or may not be connected, i.e., "so close" to one another, that there can be no other parts "inserted in between".

166. **Connector:** We shall here, by a connector, mean a hardware, or some software device that "connects" two like devices, hardware+hardware, or software+software. (Typically, in software engineering, when "connecting" two independently developed *component*[131]s, one deploys a connector in order to connect them.)

167. **Connective:** By a connective is here meant one of the Boolean "operators": "and" $\wedge$, "or" $\vee$, "imply" $\Rightarrow$, and "negation" $\sim$.

168. **Consistent:** A set of *axiom*[75]s is said to be consistent if, by means of these, and some *deduction rule*[206]s, one cannot *prove* a property and its negation.

169. **Consistency:** Being *consistent*[168] (throughout).

170. **Constraint:** By a constraint we shall here, in a somewhat narrow sense, understand a property that must be satisfied by certain values of a given type. (That is: The type may define more values than are to be satisfied by the constraint. We also use the terms *data invariant*[196], or *well-formedness*[812]. The term constraint has taken on a larger meaning than propagated in this book. We refer to *constraint programming, constraint satisfaction problems*, etc. For a seminal text book we refer to [8]. In constraint programming a constraint, as expressed in a problem model, and hence in a constraint program, is a relation on a sequence of values of (a sequence of) variables of that program.
As you see, the difference, in the two meanings of 'constraint', really, is minor.)

171. **Constructor:** By a constructor we mean either of two, albeit related, things, a type constructor, or a value constructor. By a type constructor we mean an operator on types which when applied to types, say A, constructs another type, say B. By a value constructor we mean a sometimes distributed fix operator which when applied to one or more values constructs a value of a different type. (Examples of type constructors are **-set**, $\times$, *, $^\omega$, $\overrightarrow{m}$, $\rightarrow$, $\xrightarrow{\sim}$ (sets, Cartesians, finite lists, finite and infinite lists, maps, total functions, partial functions), and mk_B. Examples of value constructors are: {•,•,...,•}, (•,•,...,•), ⟨•,•,...,•⟩, [•↦•,•↦•,...,•↦•] and mk_B(•,•,...,•), etc., (sets, Cartesians, lists, maps, and variant records).)

172. **Context:** There are two related meanings: (i) the parts of a discourse that surround some text and (ii) the interrelated conditions in which something is understood. (The former meaning emphasises *syntactical* properties, i.e., speaks of a syntactic context; the latter, we claim, *semantical* properties (i.e., semantic context). We shall often, by a syntactic context speak of the *scope*[649] of an *identifier*[351]: the text (parts) over which the identifier is defined, i.e., is *bound*[95]. And by a semantic context we then speak of the *environment*[275] in which an *identifier*[351] is *bound*[95] to its semantic meaning. As such semantic contexts go, hand-in-hand, in *configuration*[160]s, with *state*[705]s.)

173. **Context-Free:** By context-free we mean that something is defined free of any considerations of the *context*[172] in which that "something" (otherwise) occurs. (We shall use the context-free concept extensively: *context-free grammar*[175] and *context-free syntax*[176], etc. The *type definition*[785] *rule*[638]s of RSL have a context-free interpretation.)

174. **Context-Free language:** By a context-free language we mean a *language*[417] which can be generated by a *context-free syntax*[176]. (See *generator*[322].)

175. **Context-Free Grammar:** See *context-free syntax*.

176. **Context-Free Syntax:** By a context-free syntax we shall understand a type system consisting of type definitions in which right-hand-side occurrences of defined *type name*[787]s can be freely substituted for any of a variety of their definitions. (Typically a *BNF Grammar*[92] specifies a context-free syntax.)

177. **Context-Sensitive Grammar:** See *context-sensitive syntax*.

178. **Context-Sensitive Syntax:** By a context-sensitive syntax we may understand a type system consisting of ordinary type definitions in which right-hand-side occurrences of defined *type name*[787]s cannot be freely substituted for any of a variety of their definitions, but may only be substituted provided these right-hand-side type names (i.e., *nonterminal*[484]s) occur in specified contexts (of other type names or *literal*[429]s). (Usually a context-sensitive syntax can be specified by a set of rules where both left-hand and right-hand sides are composite type expressions. The left-hand-side composite expression then specifies the contexts in which the right-hand side may be substituted.)

179. **Continuation:** By a continuation we shall, rather technically, understand a state-to-state transformation function, specifically one that is the denotation of a *program point*[548], that is, of any computation as from that program point (i.e., *label*[410]) onwards — until program *termination*[751].

180. **Continuous:** Of a mathematical curve, i.e., function: 'Having the property that the absolute value of the numerical difference between the value at a given point and the value at any point in a neighborhood of the given point can be made as close to zero as desired by choosing the neighborhood small enough' [214].

181. **Contract:** A contract is a *script*[651] specifically expressing a legally binding agreement between two or more parties — hence a document describing the conditions of the contract; a contract is business arrangement for the supply of goods or services at fixed prices, times and locations. In software development a contract specifies what is to be developed (a *domain description*[243], a *requirements prescription*[615], or a *software design*[688]), how it might, or must be developed, criteria for acceptance of what has been developed, delivery dates for the developed items, who the "parties" to the contract are: the *client*[116] and the *developer*[227], etc.

    A legally binding agreement between two or more parties — hence a document describing the conditions of the contract.

    In domains a contract is a set of *rule*[638]s and *regulation*[595]s.

182. **Control:** To control has two meanings: to check, test or verify by evidence or experiments, and to exercise restraining or directing influence over, to regulate. (We shall mostly mean the second form. And we shall often use the term 'control' in conjunction with the term '*monitor*[466]ing'.)

183. **Controller:** By a controller we here mean a *computing system*[151], which interfaces with some physical environment, a *reactive*[578] system, i.e., a plant, and which, by temporally sensing (i.e., sampling) characteristic values of that plant, and by similarly regularly activating *actuator*[17]s in the plant, can make the plant behave according to desired prescriptions. (We stress the reactive system nature of the plant to be controlled. See also *sensor*[659].)

184. **Conversion:** By conversion we shall here, in a rather limiting sense, with a base in the *Lambda-calculus*[412], understand either an *Alpha-renaming*[35] or a *Beta-reduction*[84] of some *Lambda-expression*[414]. (We refer to Chap. 7.)

185. **Correct:** See next entry: *correctness*[186].

186. **Correctness:** Correctness is a relation between two specifications A and B: B is correct with respect to A if every property of what is specified in A is a property of B. (Compare to *conformance*[161] and *congruence*[163].)

187. **Corrective maintenance:** By corrective maintenance we understand a change, predicated by a specification A, to a specification, B', resulting in a specification, B'', such that B'' satisfies more properties of A than does B'. (That is: Specification B' is in error in that it is not *correct*[185] with respect to A. But B'' is an improvement over B'. Hopefully B'' is then correct wrt. A. We also refer to *adaptive maintenance*[21], *perfective maintenance*[519], and *preventive maintenance*[541].)

188. **CSP:** Abbreviation for Communicating Sequential Processes. (See [130, 203] and Chap. 21. Also, but not in this book, a term that covers constraint satisfaction problem (or programming).)

189. **Curry:** Name of American mathematician: Haskell B. Curry. Also a verb: to Curry — see *Currying*[191].

190. **Curried:** A *function invocation*[317], commonly written $f(a_1, a_2, ..., a_n)$, is said to be Curried when instead written: $f(a_1)(a_2)...(a_n)$. (The act of rewriting a function invocation into Curried form is called *Currying*[191].)

191. **Currying:** A *function signature*[318], normally written, f: A×B×...×C→D can be Curried into being written f: A→B→...→C→D. The act of doing so is called Currying.

192. **Customer:** By a customer we mean either of three things: (i) the *client*[116], a person, or a company, which orders the development of some software, or (ii) a *client*[116] *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., the *server*[663]), in order to have that server perform some *action*[12]s on behalf of the client, or (iii) a user of some software (i.e., computing system). (We shall normally use the term customer in the third sense (iii).)

. . . . . $\mathcal{D}$

193. **Data:** Data is formalised representation of information. (In our context information is what we may know, informally, and even express, in words, or informal text or diagrams, etc. Data is correspondingly the internal computer, including database representation of such information.)

194. **Data abstraction:** Data abstraction takes place when we abstract from the particular formal representation of data.

195. **Database:** By a database we shall generally understand a large collection of data. More specifically we shall, by a database, imply that the data are organised according to certain data structuring and data *query*[571] and *update*[794] principles. (Classically, three forms of (data structured) databases can be identified: The *hierarchical*[339], the *network*[478], and the *relation*[599]al database forms. We refer to [75, 76] for seminal coverage, and to [29, 28, 54, 55] for formalisation, of these database forms.)

196. **Data invariant:** By a *data*[193] invariant is understood some property that is expected to hold for all instances of the data. (We use the term 'data' colloquially, and really should say type invariance, or variable content invariance. Then 'instances' can be equated with values. See also *constraint*[170].)

197. **Data refinement:** Data refinement is a relation. It holds between a pair of data if one can be said to be a "more concrete" implementation of the other. (The whole point of *data abstraction*[194], in earlier *phase*[523]s, *stage*[702]s and *step*[711]s of *development*[228], is that we can later concretise, i.e., data refine.)

198. **Data reification:** Same as *data refinement*[197]. (To reify is to render something abstract as a material or concrete thing.)

199. **Data structure:** By a data structure we shall normally understand a composition of *data*[193] *value*[802]s, for example, in the "believed" form of a linked *list*[428], a *tree*[777], a *graph*[327] or the like. (As in contrast to an *information structure*[374], a data structure (by our using the term *data*[193]) is bound to some computer representation.)

200. **Data type:** By a *data*[193] *type*[782] is understood a set of *value*[802]s and a set of *function*[310]s over these values — whether *abstract*[1] or *concrete*[154].

201. **Declaration:** A declaration prescribes the allocation of a resource of the kind declared: (i) A variable, i.e., a location in some storage; (ii) a channel between active processes; (iii) an object, i.e., a process possessing a local state; etc.

202. **Decidable:** A formal logic system is decidable if there is an *algorithm*[31] which prescribes *computation*[144]s that can determine whether any given sentence in the system is a theorem.

203. **Decomposition:** By a decomposition is meant the presentation of the parts of a *composite*[133] "thing".

204. **Deduce:** To perform a *deduction*[205], see next. (Cf. *infer*[368].)

205. **Deduction:** A form of reasoning where a conclusion about particulars follows from general premises. (Thus deduction goes from the general (case) to the specific (case). See contrast to *induction*[364]: inferring from specific cases to general cases.)

206. **Deduction rule:** A *rule*[638] for performing *deduction*[205]s.

207. **Definiendum:** The left-hand side of a *definition*[210], that which is to be defined.

208. **Definiens:** The right-hand side of a *definition*[210], that which is defining "something".

209. **Definite:** Something which has specified limits. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

210. **Definition:** A definition defines something, makes it conceptually "manifest". A definition consists of two parts: a *definiendum*[207], normally considered the left-hand part of a definition, and a *definiens*[208], normally considered the right-hand part (the body) of a definition.

211. **Definition set:** By a definition set we mean, given a *function*[310], the set of *value*[802]s for which the function is defined, i.e., for which, when it is *applied* to a member of the definition set yields a proper value. (Cf., *range set*[577].)

212. **Delimiter:** A delimiter delimits something: marks the start, and/or end of that thing. (A delimiter thus is a syntactic notion.)

213. **Denotation:** A direct specific meaning as distinct from an implied or associated idea [214]. (By a denotation we shall, in our context, associate the idea of mathematical functions: That is, of the *denotational semantics*[215] standing for functions.)

214. **Denotational:** Being a *denotation*[213].

215. **Denotational semantics:** By a denotational semantics we mean a *semantics*[655] which to *atomic*[63] syntactical notions associate simple mathematical structures (usually *function*[310]s, or *set*[664]s of *trace*[766]s, or *algebra*[26]s), and which to *composite*[133] syntactical notions prescribe a semantics which is the *functional*[312] *composition*[140] of the denotational semantics of the *composition*[140] parts.

216. **Denote:** Designates a mathematical meaning according to the principles of *denotational semantics*[215]. (Sometimes we use the looser term designate.)

217. **Dependability:** Dependability is defined as the property of a *machine*[436] such that reliance can justifiably be placed on the service it delivers [197]. (See definition of the related terms: *error*[278], *failure*[286], *fault*[287] and *machine service*[439].)

218. **Dependability requirements:** By *requirements*[605] concerning dependability we mean any such requirements which deal with either *accessibility*[8] requirements, or *availability*[74] requirements, or *integrity*[388] requirements, or *reliability*[601] requirements, or *robustness*[631] requirements, or *safety*[642] requirements, or *security* requirements.

219. **Describe:** To describe something is to create, in the mind of the reader, a *model*[460] of that something. The thing, to be describable, must be either a physically manifest *phenomenon*[524], or a concept derived from such phenomena. Furthermore, to be describable it must be possible to create, to formulate a mathematical, i.e., a formal description of that something. (This delineation of description is narrow. It is too narrow for, for example, philosophical or literary, or historical, or psychological discourse. But it is probably too wide for a *software engineering*[693], or a *computing science*[150] discourse. See also *description*[220].)

220. **Description:** By a description is, in our context, meant some text which designates something, i.e., for which, eventually, a mathematical *model*[460] can be established. (We readily accept that our characterisation of the term 'description' is narrow. That is: We take as a guiding principle, as a dogma, that an informal text, a *rough sketch*[634], a *narrative*[476], is not a description unless one can eventually demonstrate a mathematical model that somehow relates to, i.e., "models" that informal text. To further paraphrase our concern about "describability", we now state that a description is a description of the *entities*, *function*[310]s, *event*[281]s and *behaviour*[79]s of a further designated universe of discourse: That is, a description of a *domain*[239], a *prescription*[540] of *requirements*[605], or a *specification*[698] of a *software design*[688].)

221. **Design:** By a design we mean the *specification*[698] of a *concrete*[154] *artefact*[55], something that can either be physically manifested, like a chair, or conceptually demonstrated, like a software program.

222. **Designate:** To designate is to present a reference to, to point out, something. (See also *denote*[216] and *designation*[223].)

223. **Designation:** The relation between a *syntactic* marker and the semantic thing signified. (See also *denote*[216] and *designate*[222].)

224. **Destructor:** By a destructor we shall here understand a *function*[310] which applies to a *composite*[133] *value*[802] and yields a further specified part (i.e., a subpart) of that value. (Examples of destructors in RSL are the list indexing function, and the selector functions of a variant record. They do not destroy anything, however.)

225. **Determinate:** ()

226. **Deterministic:** In a narrow sense we shall say that a behaviour, a process, a set of actions, is deterministic if the outcome of the behaviour, etc., can be predicted: Is always the same given the same "starting conditions", i.e., the same initial *configuration*[160] (from which the behaviour, etc., proceeds). (See also *nondeterministic*[481].)

227. **Developer:** The person, or the company, which constructs an *artefact*[55], as here, a *domain description*[243], or a *requirements prescription*[615], or a *software design*[688].

228. **Development:** The set of actions that are carried out in order to construct an *artefact*[55].

229. **Diagram:** A usually two-dimensional drawing, a figure. (Sometimes a diagram is annotated with informal and *formal*[296] text.)

230. **Dialogue:** A "conversation" between two *agent*[24]s (men or machines). (We thus speak of man-machine dialogues as carried out over *CHI*[112]s (*HCI*[334]s).)

231. **Didactics:** Systematic instruction based on a clear conceptualisation of the bases, of the foundations, upon which what is being instructed rests. (One may speak of the didactics of a field of knowledge, such as, for example, software engineering. We believe that the present three volume book represents such a clearly conceptualised didactics, i.e., a foundationally consistent and complete basis.)

232. **Directed graph:** A directed graph is a *graph*[327] all of whose *edge*[262]s are directed, i.e., are *arrow*[54]s.

233. **Directory:** A collection of directions. (We shall here take the more limited view of a directory as being a list of names of, i.e., references to *resource*[620]s.)

234. **Discharge:** We use the term discharge in a very narrow sense, namely that of discharging a proof obligation, i.e., by carrying out a proof.

235. **Discrete:** As opposed to *continuous*[180]: consisting of distinct or unconnected elements [214].

236. **Disjunction:** Being separated, being disjoined, decomposed. (We shall mostly think of disjunction as the (meaning of the) logical connective "or": $\vee$.)

237. **Document:** By a document is meant any text, whether informal or *formal*[296], whether *informative*, *descriptive* (or *prescriptive*) or *analytic*[40]. (Descriptive documents may be *rough sketch*[634]es, *terminologies*, *narrative*[476]s, or *formal*[296]. Informative documents are not *descriptive*. Analytic documents "describe" relations between documents, *verification*[807] and *validation*[800], or describe properties of a document.)

238. **Documentation requirements:** By documentation requirements we mean requirements which state which kinds of documents shall make up the deliverable, what these documents shall contain and how they express what they contain.

239. **Domain:** Same as *application domain*[46]; hence see that term for a characterisation. (The term domain is the preferred term.)

240. **Domain acquisition:** The act of acquiring, of gathering, *domain knowledge*[254], and of analysing and recording this knowledge.

241. **Domain analysis:** The act of analysing recorded *domain knowledge*[254] in search of (common) properties of phenomena, or relating what may be considered separate phenomena.

242. **Domain capture:** The act of gathering *domain knowledge*[254], of collecting it — usually from domain *stakeholder*[703]s.

243. **Domain description:** A textual, informal or formal document which describes a domain **as it is**. (Usually a domain description is a set of documents with many parts recording many facets of the domain: The *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], *rules and regulations*[640], and the *human behaviour*[345]s.)

244. **Domain description unit:** By a domain description unit we understand a short, "one- or two-liner", possibly *rough-sketch*[633] *description*[220] of some property of a *domain*[239] *phenomenon*[524], i.e., some property of an *entity*[272], some property of a *function*[310], of an *event*[281], or some property of a *behaviour*[79]. (Usually domain description units are the smallest textual, sentential fragments elicited from domain *stakeholder*[703]s.)

245. **Domain determination:** Domain determination is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] cum *requirements prescription*[615]. Any *nondeterminism*[482] expressed by either of these specifications which is not desirable for some required software design must be made deterministic (by this *requirements engineer*[612] performed operation). Other domain requirements facets are: *domain projection*[255], *domain instantiation*[253], *domain extension*[249] and *domain fitting*[251].

246. **Domain development:** By domain development we shall understand the *development*[228] of a *domain description*[243]. (All aspects are included in development: *domain acquisition*[240], domain *analysis*[39], domain *model*[460]ling, domain *validation*[800] and domain *verification*[807].)

247. **Domain engineer:** A domain engineer is a *software engineer*[692] who performs *domain engineering*[248]. (Other forms of *software engineer*[692]s are: *requirements engineer*[612]s and *software design*[688]ers (cum *programmer*[547]s).)

248. **Domain engineering:** The engineering of the development of a *domain description*[243], from identification of *domain*[239] *stakeholder*[703]s, via *domain acquisition*[240], *domain analysis*[241] and *domain description*[243] to *domain validation*[256] and *domain verification*[257].

249. **Domain extension:** Domain extension is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] or a *requirements prescription*[615]. It effectively extends a *domain description*[243] by entities, functions, events and/or behaviours conceptually possible, but not necessarily humanly or technologically feasible in the domain (as it was). Other domain requirements facets are: *domain projection*[255], *domain determination*[245], *domain instantiation*[253] and *domain fitting*[251].

250. **Domain facet:** By a domain facet we understand one amongst a finite set of generic ways of analysing a domain: A view of the domain, such that the different facets cover conceptually different views, and such that these views together cover the domain. (We consider here the following domain facets: *business process*[99], *intrinsics*[399], *support technology*[725], *management and organisation*[445], *rules and regulations*[640], and *human behaviour*[345].)

251. **Domain fitting:** By *domain requirements fitting* we understand an operation which takes n domain requirements prescriptions, d_{r_i}, that are claimed to share m independent sets of tightly related sets of simple entities, actions, events and/or behaviours and map these into $n+m$ domain requirements prescriptions, δ_{r_j}, where m of these, $\delta_{r_{n+k}}$ capture the shared phenomena and concepts and the other n prescriptions, δ_{r_ℓ}, are like the n "input" domain requirements prescriptions, d_{r_i}, except that they now, instead of the "more-or-less" shared prescriptions, that are now consolidated in $\delta_{r_{n+k}}$, prescribe interfaces between δ_{r_i} and $\delta_{r_{n+k}}$ for $i : \{1..n\}$. Other domain requirements facets are: *domain projection*[255], *domain determination*[245], *domain instantiation*[253] and *domain extension*[249].

252. **Domain initialisation:** Domain initialisation is an *interface requirements facet*[395]. It is an operation performed on a *requirements prescription*[615]. For an explanation see *shared data initialisation*[670] (its 'equivalent'). Other *interface requirements facet*[395]s are: *shared data refreshment*[672], *computational data+control*[146], *man-machine dialogue*[446], *man-machine physiological*[448] and *machine-machine dialogue*[437] *requirements*[605].

253. **Domain instantiation:** Domain instantiation is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] (cum *requirements prescription*[615]). Where, in a domain description certain *entities* and *function*[310]s are left undefined, domain instantiation means that these entities or functions are now instantiated into constant *value*[802]s. Other requirements facets are: *domain projection*[255], *domain determination*[245], *domain extension*[249] and *domain fitting*[251].

254. **Domain knowledge:** By domain knowledge we mean that which a particular group of people, all basically engaged in the "same kind of activities", know about that domain of activity, and what they believe that other people know and believe about the same domain. (We shall, in our context, strictly limit ourselves to "knowledge", staying short of "beliefs", and we shall similarly strictly limit ourselves to assume just one "actual" world, not any number of "possible" worlds. More specifically, we

shall strictly limit our treatment of domain knowledge to stay clear of the (albeit very exciting) area of reasoning about knowledge and belief between people (and agents) [127, 106].)

255. **Domain projection:** Domain projection is a *domain requirements facet*[259]. It is an operation performed on a *domain description*[243] cum *requirements prescription*[615]. The operation basically "removes" from a description definitions of those *entities* (including their *type definition*[785]s), *functions*, *events* and *behaviours* that are not to be considered in the *requirements*[605]. The removed phenomena and concepts are thus projected "away". Other domain requirements facets are: *domain determination*[245], *domain instantiation*[253], *domain extension*[249] and *domain fitting*[251].

256. **Domain validation:** By domain validation we rather mean: '*validation*[800] of a domain description', and by that we mean the informal assurance that a description purported to cover the *entities*, *function*[310]s, *event*[281]s and *behaviour*[79]s of a further designated domain indeed does cover that domain in a reasonably representative manner. (Domain validation is, necessarily, an informal activity: It basically involves a guided reading of a domain description (being validated) by *stakeholder*[703]s of the domain, and ends in an evaluation report written by these domain *stakeholder*[703] readers.)

257. **Domain verification:** By domain verification we mean *verification*[807] of claimed properties of a domain description, and by that we mean the formal assurance that a description indeed does possess those claimed properties. (The usual principles, techniques and tools of verification apply here.)

258. **Domain requirements:** By domain *requirements*[605] we understand such requirements — save those of *business process reengineering*[101] — which can be expressed sôlely by using professional terms of the *domain*[239]. (Domain requirements constitute one requirements *facet*[285]. Others requirements facets are: *business process reengineering*[101], *interface requirements*[394] and *machine requirements*[438].)

259. **Domain requirements facet:** By *domain requirements*[258] facets we understand such domain requirements that basically arise from either of the following operations on *domain description*[243]s (cum *requirements prescription*[615]s): *domain projection*[255], *domain determination*[245], *domain extension*[249], *domain instantiation*[253] and *domain fitting*[251].

260. **Dynamic:** An *entity*[272] is said to be dynamic if its value changes over time, i.e., it is subjected, somehow, to actions. (We distinguish three kinds of dynamic entities: *inert*[367], *active*[14] and *reactive*[578]. This is in contrast to *static*[708].)

261. **Dynamic typing:** Enforcement of *type checking* at *run time*[641]. (A language is said to be dynamically typed if it is not *statically typed*.)

. . . . . $\mathcal{E}$

262. **Edge:** A line, a connection, between two *node*[479]s of a *graph*[327] or a *tree*[777]. (Other terms for the same idea are: *arc*[50] and *branch*[97].)

263. **Elaborate:** See next: *elaboration*[264].

264. **Elaboration:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term elaboration in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to pairs of states and values.

265. **Elicitation:** To elicit, to extract. (See also: *acquisition*[11]. We consider elicitation to be part of acquisition. Acquisition is more than elicitation. Elicitation, to us, is primarily the act of extracting information, i.e., knowledge. Acquisition is that plus more: Namely the preparation of what and how to elicit and the postprocessing of that which has been elicited — in preparation of proper analysis. Elicitation applies both to domain and to requirements elicitation.)

266. **Embedded:** Being an integral part of something else. (When something is embedded in something else, then that something else is said to surround the embedded thing.)

267. **Embedded system:** A *system*[736] which is an integral part of a larger system. (We shall use the term embedded system primarily in the context of the larger, 'surrounding' system being *reactive*[578] and/or *hard real time*[330].)

268. **Endomorphism:** A *homomorphism*[343] that maps an algebra into itself is an endomorphism. (See also *automorphism*[72], *epimorphism*[276], *isomorphism*[404], *monomorphism*[467].)

269. **Engineer:** An engineer is a person who "walks the bridge" between science and technology: (i) Constructing, i.e., designing, *technology*[746] based on scientific insight, and (ii) analysing technology for its possible scientific content.

270. **Engineering:** Engineering is the design of *technology*[746] based on scientific insight, and the analysis of technology for its possible scientific content. (In the context of this glossary we single out three forms of engineering: *domain engineering*[248], *requirements engineering*[613] and *software design*[688]; together we call them *software engineering*[693]. The technology constructed by the *domain engineer*[247] is a *domain description*[243]. The technology constructed by the *requirements engineer*[612] is a *requirements prescription*[615]. The technology constructed by the *software design*[688]er is *software*[685].)

271. **Enrichment:** The addition of a property to something already existing. (We shall use the term enrich in connection with a collection (i.e., a RSL **scheme** or a RSL

class) — of definitions, declaration and axioms — being '**extend**ed **with**' further such definitions, declaration and axioms.)

272. **Entity:** By an entity we shall understand either a *simple entity*[681], an *action*[12], an *event*[281] or a *behaviour*[79].

273. **Enumerable:** By enumerable we mean that a set of elements satisfies a *proposition*[560], i.e., can be logically characterised.

274. **Enumeration:** To list, one after another. (We shall use the term enumeration in connection with the syntactic expression of a "small", i.e., definite, number of elements of a(n enumerated) *set*[664], *list*[428] or *map*[449].)

275. **Environment:** A context, that is, in our case (i.e., usage), the ("more static") part of a *configuration*[160] in which some syntactic entity is *elaborated*, *evaluated*, or *interpreted*. (In our "metacontext", i.e., that of software engineering, environments, when deployed in the elaboration (etc.) of, typically, specifications or programs, record, i.e., list, associate, identifiers of the specification or program text with their meaning.)

276. **Epimorphism:** If a *homomorphism*[343] ϕ is a *surjective function*[727] then ϕ is an epimorphism. (See also *automorphism*[72], *endomorphism*[268], *isomorphism*[404], *monomorphism*[467].)

277. **Epistemology:** The study of knowledge. (Contrast, please, to *ontology*[492].)

278. **Error:** An error is an action that produces an incorrect result. An error is that part of a *machine*[436] *state*[705] which is "liable to lead to subsequent failure". An error affecting the *machine service*[439] is an indication that a *failure*[286] occurs or has occurred [197]. (An error is caused by a *fault*[287].)

279. **Evaluate:** See next: *evaluation*[280].

280. **Evaluation:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term evaluation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to values.

281. **Event:** Something that occurs instantaneously. (We shall, in our context, take events as being manifested by certain *state*[705] changes, and by certain *interaction*[392]s between *behaviour*[79]s or *process*[544]es. The occurrence of events may "trigger" actions. How the triggering, i.e., the *invocation*[402] of *functions* are brought about is usually left implied, or unspecified. We consider *event*[281]s to be one of the four kinds of

entities[272] that the `Triptych` "repeatedly" considers. The other three are: *simple entities*[681], *action*[12]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.)

282. **Expression:** An expression, in our context (i.e., that of software engineering), is a syntactical entity which, through *evaluation*[280], designates a *value*[802].

283. **Extension:** We shall here take extension to be the same as *enrichment*[271]. In *domain requirements*[258], when we 'perform' extension, we introduce *entities*[272] (*simple entities*[681], *action*[12]s, *event*[281]s and *behaviour*[79]s) that were not [originally] in the domain [but will now become entities of the domain resulting from implementing the requirements].

284. **Extensional:** Concerned with objective reality [214]. (Please observe a shift here: We do not understand the term extensional as 'relating to, or marked by extension in the above sense, but in contrast to *intensional*[390].)

$\mathcal{F}$

285. **Facet:** By a facet we understand one amongst a finite set of generic ways of analysing and presenting a *domain*[239], a *requirements*[605] or a *software design*[688]: a view of the universe of discourse, such that the different facets cover conceptually different views, and such that these views together cover that universe of discourse. (Examples of domain facets are *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], *rules and regulations*[640] and *human behaviour*[345]. Examples of requirements facets are *business process reengineering*[101], *domain requirements*[258], *interface requirements*[394] and *machine requirements*[438]. Examples of software design facets are *software architecture*[687], *component design*[132], *module design*[465], etc.)

286. **Failure:** A *fault*[287] may result in a failure. A *machine*[436] failure occurs when the delivered *machine service*[439] deviates from fulfilling the machine function, the latter being what the machine is aimed at [197]. (A failure is thus something relative to a *specification*[698], and is due to a *fault*[287]. Failures are concerned with such things as *accessibility*[8], *availability*[74], *reliability*[601], *safety*[642] and *security*.)

287. **Fault:** The adjudged (i.e., the 'so judged') or hypothesised cause of an *error*[278] [197]. (An *error*[278] is caused by a fault, i.e., faults cause errors. A software fault is the consequence of a human *error*[278] in the development of that software.)

288. **Finite:** Of a fixed number less than infinity, or of a fixed structure that does not "flow" into perpetuity as would any *information structure*[374] that just goes on and on. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

289. **Finite state automaton:** By a finite state automaton we understand an *automaton*[71] whose state set is finite. (We shall usually consider only what is known as Moore automata: that is, automata which have some final states.)

290. **Finite state machine:** By a finite state machine we understand an extended *finite state automaton*[289]. The extension amounts simply to the following: Every transition (caused by an input, in a state, to another state) also yields an output. (We shall thus consider only what is known as Mealy machines. The output is intended to designate some action, or some signal, to be considered by an environment of the machine.)

291. **Finite state transducer:** By a finite state transducer we simply mean the same as a finite state machine. (The machine in question is said to transduce, to "translate" any sequence of inputs to some corresponding sequence of outputs.)

292. **First-order:** We say that a *predicate logic*[537] is first order when quantified variables are not allowed to range over functions. (If they range over functions we call the logic a *higher-order*[341] logic [191, 180]. Similar remarks can be made for general first-order functions, respectively higher-order functions.)

293. **Fix point:** The fix point of a function, F, is any value, f, for which $Ff = f$. A function may have any number of fixed points from none (e.g., $Fx = x+1$) to infinitely many (e.g., $Fx = x$). The fixed point combinator, written as either "**fix**" or "**Y**" will return the fixed point of a function. (The fix point identity is $\mathbf{Y}F = F(\mathbf{Y}F)$.)

294. **Fitting:** Fitting in the context of requirements engineering is an operation that applies to n (where n is 2 or more) domain requirements descriptions $(d_1, d_2, \ldots, d_n)$ and yields $n + 1$ domain requirements descriptions $(d'_1, d'_2, \ldots, d'_n$ and d"shared"$)$ where n of these each, d'_i, cover major parts of respective d_i and where d"shared" covers what is "somehow" common to $d_1, d_2, \ldots, d_n$.

295. **Flowchart:** A diagram (a chart), for example of circles (input, output), annotated (square) boxes, annotated diamonds and infixed arrows, that shows step by step flow through an algorithm.

296. **Formal:** By formal we shall, in our context (i.e., that of software engineering), mean a language, a system, an argument (a way of reasoning), a program or a specification whose syntax and semantics is based on (rules of) mathematics (including mathematical logic).

297. **Formal definition:** Same as *formal description*[299], *formal prescription*[303] or *formal specification*[304].

298. **Formal development:** Same as the standard meaning of the composition of *formal*[296] and *development*[228]. (We usually speak of a spectrum of development modes: *systematic development*[737], *rigorous development*[629], and formal development. Formal software development, to us, is at the "formalistic" extreme of the three modes of development: Complete *formal specification*[304]s are always constructed, for all (phases and) stages of development; all *proof obligation*[555]s are expressed; and all are discharged (i.e., proved to hold).)

299. **Formal description:** A *formal*[296] *description*[220] of something. (Usually we use the term formal description only in connection with *formalisation*[300] of *domain*[239]s.)

300. **Formalisation:** The act of making a formal specification of something elsewhere informally specified; or the document which results therefrom.

301. **Formal method:** By a formal method we mean a *method*[456] whose techniques and tools[14] are *formal*[296]ly based. (It is common to hear that some notation is claimed to be that of a formal method — where it then turns out that few, if any, of the building blocks of that notation have any formal foundation. This is especially true of many diagrammatic notations. UML is a case in point — much is presently being done to formalise subsets of UML [181].)

302. **Formal parameter:** By a formal parameter we mean an identification (say a naming and a typing), in a *function definition*[316]'s *function signature*[318], of an argument of the function, a place-holder for *actual argument*[16]s.

303. **Formal prescription:** Same as *formal definition*[297] or *formal specification*[304]. (Usually we use the term formal prescription only in connection with *formalisation*[300] of *requirements*[605].)

304. **Formal specification:** A *formalisation*[300] of something. (Same as *formal definition*[297], *formal description*[299] or *formal prescription*[303]. Usually we use the term formal specification only in connection with *formalisation*[300] of *software design*[688]s.)

305. **Free:** The concept of being free is associated with (i) *identifier*[351]s (i.e., *name*[474]s) and *expression*[282]s, and (ii) with *name*[474]s (i.e., *identifier*[351]s) and *resource*[620]s. An identifier is said to be either *bound*[95] or free in an expression based on certain rules being satisfied or not. If an identifier is free in an expression then nothing is said about what free occurrences of that identifier are bound to. (Cf. *bound*[95].)

306. **Freeing:** The removal of *storage*[715] *location*[431]s, or of *stack activation*[701]s.

307. **Frontier:** The concept of frontier is here associated with *tree*[777]s. Visualise that tree as represented as a flat diagram with no crosses (i.e., intersecting) *branch*[97]es. A frontier of a tree is a reading of the leaves (cf. *leaf*[419]) of the tree in one of the two possible directions, say left to right or right to left. (See *tree traversal*[778].)

308. **FUNARG:** A specification or a programming language is said to enjoy, i.e. possess, the FUNARG property if *value*[802]s of *function invocation*[317]s may be *function*[310]s defined locally to the invoked function. (LISP has the FUNARG property. So does SAL, a simple applicative language defined in .)

[14] Tools include specification and programming languages as such, as well as all the software tools relating to these languages (editors, syntax checkers, theorem provers, proof assistants, model checkers, specification and program (flow) analysers, interpreters, compilers, etc.).

309. **Full algebra:** A full *algebra* is a *total algebra*[765].

310. **Function:** By a function we understand something which when *applied* to a *value*[802], called an *argument*, yields a value called a *result*. (Functions can be modelled as sets of (argument, result) pair — in which case applying a function to an argument amounts to "searching" for an appropriate pair. If several such pairs have the same argument (value), the function is said to be *nondeterministic*[481]. If a function is applied to an argument for which there is no appropriate pair, then the function is said to be partial; otherwise it is a total function.)

311. **Function activation:** When, in an operational, i.e., computational ("mechanical") sense, a function is being applied, then some resources have to be set aside in order to carry out, to handle, the application. This is what we shall call a function activation. (Typically a function activation, for conventional *block-structured*[90] languages (like `C#, Java, Standard ML` [125, 208, 119]), is implemented by means (also) of a stack-like data structure: Function invocation then implies the stacking (pushing) of a stack activation on that stack, i.e., the *activation stack*[13] (a circular reference!). Elaboration of the function definition body means that intermediate values are pushed and popped from the topmost activation element, etc., and that completion of the function application means that the top stack activation is popped.)

312. **Functional:** A function whose arguments are allowed themselves to be functions is called a functional. (The *fix point*[293] (finding) function is a functional.)

313. **Functional programming:** By functional programming we mean the same as *applicative programming*[48]: In its barest rendition functional programming involves just three things: definition of functions, functions as ordinary *value*[802]s, and *function application*[315] (i.e., *function invocation*[317]). (Most current functional programming languages (`Haskell, Miranda, Standard ML`) go well beyond just providing the three basic building blocks of functional programming [220, 221, 176].)

314. **Functional programming language:** By a functional programming language we mean a *programming language*[551] whose principal values are functions and whose principal operations on these values are their creation (i.e., definition), their application (i.e., invocation) and their composition. (Functional programming languages of interest today, 2005, are (alphabetically listed): `CAML` [69, 63, 64, 227, 157], `Haskell` [220], `Miranda` [221], `Scheme` [1, 115, 97] and `SML` (Standard ML) [176, 119]. `LISP 1.5` was a first functional programming language [170].)

315. **Function application:** The act of applying a function to an argument is called a function application. (See 'comment' field of *function activation*[311] just above.)

316. **Function definition:** A *function*[310] *definition*[210], as does any definition, consists of a *definiens*[208] and a *definiendum*[207]. The definiens is a *function signature*[318] and the definiendum is a clause, typically an expression. (Cf. *Lambda-function*[415]s.)

317. **Function invocation:** Same as *function application*[315]. (See parenthesized remark of entry 311 (*function activation*[311]).)

318. **Function signature:** By a function signature we mean a text which presents the name of the function, the types of its argument values and the type(s) of its result value(s).

. . . . . $\mathcal{G}$

319. **Garbage:** By garbage we shall here understand those (computing) *resource*[620]s which can no longer be referenced. (Usually we restrict our 'garbage' concern to that of *storage*[715] *location*[431]s that can no longer be accessed because there are no references to them.)

320. **Garbage collector:** To speak of garbage collection we must first introduce the notions of allocatable *storage*[715], i.e., storage — what shall be known as free, i.e., unallocated — *location*[431]s (including those that can be considered *garbage*[319]). By a garbage collector we shall here understand a device, a software program or a hardware mechanism which "returns" to a set of free locations that can subsequently be made available for *allocation*[33].

321. **Generate:** By generate we shall understand that which can be associated both with a *grammar*[325] and with an *automaton*[71]: namely a *language*[417], i.e., a set of strings. Either accepted as *input*[382] to a *finite state automaton*[289], or *denote*[216]d by a *grammar*[325]. (Acceptance by an automaton means that the automaton is started in an initial state and upon completion of reading the input is in a final state. Generation by a grammar means the recursive (i.e., repeated) *substitution*[722] of *nonterminal*[484]s of a grammar *rule*[638] left-hand side with the left-hand sides of the rules whose right-hand side is the substituted nonterminal.)

322. **Generator:** A generator is a concept: It can be thought of as a device, i.e., a software program or a machine mechanism, which outputs typically sequences of structures — typically symbols. (A *BNF Grammar*[92] can thus be said to generate the (usually infinite) set of strings, i.e., of sentence of the designated language. A *finite state machine*[290] can likewise be said to be a generator: Upon being presented with any input string it generates an output string (a transduction).)

323. **Generator function:** To speak of a generator function we need first introduce the concept of a *sort*[694] "of interest". A generator function is a function which when applied to arguments of some kind, i.e., types, yields a value of the type of the sort "of interest". (Typically the sort "of interest" can be thought of as the state (a stack, a queue, etc.).)

324. **Glossary:** According to [160] a *gloss* is "a word inserted between the lines or in the margin as an explanatory rendering of a word in the text; hence a similar rendering in a glossary or dictionary. Also, a comment, explanation, interpretation."

Furthermore according to [160] a *glossary* is therefore "a collection of glosses, a list with explanations of abstruse, antiquated, dialectical, or technical terms; a partial dictionary."

325. **Grammar:** See *syntax*, in general, or *regular syntax*, *context-free syntax*, *context-sensitive syntax* and *BNF* in specific.

326. **Grand state:** "Grand state" is a colloquial term. It is meant to have the same meaning as *configuration*[160]. (The colloquialism is used in the context of, for example, praising a software engineer as "being one who really knows how to design the grand state for some universe of discourse" being specified.)

327. **Graph:** By a graph we shall here mean the term as usually used in the discrete mathematics discipline of graph theory: as a (usually, but not necessarily finite) set of *node*[479]s (*vertexes*), some of which may be connected by (one or more) *arc*[50]s (*edge*[262]s, lines). (A graph edge defines a *path*[517] of length one. If there is a path from one node to another, and from that other node to yet a third node, then the graph, by transitivity, defines a path from the first to the third node, etc. A graph can be either an *acyclic graph*[19] (no path "cycles back") or a *cyclic graph*, a *directed graph*[232] (edges are one-directional arrows) or an *undirected graph* [20, 21, 183, 122].)

328. **Ground term:** A ground term is either an *identifier*[351] or a *value*[802] *literal*[429]. (The identifier is then assumed to be bound to a value. The value literal typically is an alphanumeric string designating, for example, an integer, a real, a truth value, a character, etc.)

329. **Grouping:** By grouping we mean the ordered, finite collection, into a *Cartesian*[107], of mathematical structures (i.e., *value*[802]s).

. . . . . $\mathcal{H}$

330. **Hard real time:** By hard real time we mean a *real time*[580] property where the exact, i.e., absolute timing, or time interval, is of essence. (Thus, if a system is said to enjoy, or must possess, a certain real time property, for example, (i) the system must emit a certain signal on the 11th of December 2015 at 17:20:30 hours[15], or (ii) that a response signal must be issued after an interval of exactly 1234 days, 5 hours, 6 minutes, and 7 seconds plus/minus 8 microseconds (from when an initiating signal was received), then it is hard real time. Cf. *soft real time*[684].)

331. **Hardware:** By hardware is meant the physical embodiment of a computer: its electronics, its boards, the racks, cables, button, lamps, etc.

332. **Hazard:** A hazard is a source of danger.

[15] That time is when the current author hopes to celebrate the exact hour of his anniversary of 50 years of marriage to Kari Skallerud!

333. **Hazard analysis:** Hazard analysis is a process used to determine how a device can cause hazards to occur and then reducing the risks to an acceptable level. (The process consists of: (1) the developer of the system determining what could go wrong with the device, (2) determining how the effects of the failure can be mitigated, and (3) implementing and testing mitigations.)

334. **HCI:** Abbreviation for human computer interface. (Same as *CHI*[112], and same as *man-machine*[446] interface.)

335. **Heap:** By a heap is here meant an unordered, finite collection, i.e., a set, of *storage*[715] *location*[431]s, such that each of these locations can be said to be allocated (for some purpose), and such that a freeing, i.e., deallocation, of these locations usually does not follow the inverse order of their allocation. (Thus a heap works in contrast to an *activation stack*[13] — complementary, so to speak! Typically a *garbage collector*[320] is involved in helping to secure locations on the heap available for allocation.)

336. **Heterogeneous algebra:** A heterogeneous *algebra* is an algebra whose carrier A is an indexed set of carriers: A_1, A_2,..., A_m, and whose functions, $\phi_{i_n} : \Phi$, or *arity* n, are of *type*[782]: $A_{i_1} \times A_{i_2} \times \cdots \times A_{i_n} \rightarrow A_j$ where i_k, for all $k \in \{1, \ldots, n\}$, are in the set $\{1, 2, \ldots, m\}$.

337. **Hiding:** Hiding is a concept related to *module*[464]s. In fact, it is a main purpose of syntactically providing the module mechanism. You have, somewhat mechanistically, to imagine a group of (developers of) modules. One module mentions (i.e., uses), say, functions defined in other modules. But those other modules, besides, in order to define those "exported" functions, define auxiliary functions (types, etc.) that "reveal" details of implementation which it is not necessary to divulge. (One may wish, later, in "the life of that module", to change those implementation decisions.) Hence, by syntactic means, such as, for example, *export*, *import* and *hide* clauses, the developer requests the module compiling system to statically (or otherwise) secure that other modules cannot "inspect" those auxiliary functions, types, etc. (We refer to [187, 186, 190, 189, 188]. Parnas must be credited, among others, for having skillfully propagated the hiding concept.)

338. **Hierarchy:** By a hierarchy we understand a conceptual decomposition of resources into what can be "pictured" as a *tree*[777]-like structure (and where the emphasis is on the root of the structure).

339. **Hierarchical:** By something being hierarchical we mean that that something forms a *hierarchy*[338]. (See also *compositional*[141].)

340. **Hierarchical documentation:** By hierarchical documentation we mean a development, or a presentation (of that development), of, as here, some *description*[220] (*prescription*[540] or *specification*[698]), in which a notion of "largest", overall, phenomena and concepts are developed (resp. presented) first, then their decompositions into

component phenomena and concepts, etc., until some notion of atomic, i.e., "smallest" development (etc.) has been achieved. (See also *hierarchy*[338] (just above) and *compositional documentation*[142].)

341. **Higher-order:** A *functional*[312] or a *value*[802] whose *definition set*[211] or *range set*[577] values are *function*[310]s. (See, in contrast, *first-order*[292].)

342. **Homeomorphism:** A function that is a one-to-one mapping between sets such that both the function and its inverse are continuous. (Not to be confused with *homomorphism*[343].)

343. **Homomorphism:** A *function*[310], $\phi : A \to A'$, from values of the carrier A of one *algebra*[26] (A, Ω) to values of the carrier A' of another algebra (A', Ω') is said to be a homomorphism (same as a morphism) from (A, Ω) to (A', Ω'), if for any $\omega : \Omega$ and for any $a_i : A$, there is a corresponding $\omega' : \Omega'$ such that: $\phi(\omega(a_1, a_2, ..., a_n)) = \omega'(\phi(a_1), \phi(a_2), ..., \phi(a_n))$. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276], *isomorphism*[404] and *monomorphism*[467].)

344. **Homomorphic principle:** The homomorphic principle advises the software engineer to formulate *function definition*[316]s such that they express a *homomorphism*[343]. (It is a basic tenet of a *denotational semantics*[215] *definition*[210] that it is expressed as a *homomorphism*[343].)

345. **Human behaviour:** By human behaviour we shall here understand the way a human follows the enterprise *rules and regulations*[640] as well as interacts with a *machine*[436]: dutifully honouring specified (machine *dialogue*[230]) *protocol*[561]s, or negligently so, or sloppily not quite so, or even criminally not so! (Human behaviour is a *facet*[285] of the *domain*[239] (of the enterprise). We shall thus model human behaviour also in terms of it failing to react properly, i.e., humans as *nondeterministic*[481] *agent*[24]s! Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445], and *rules and regulations*[640].)

346. **Hybrid:** Something heterogeneous, something (as a computing device) that has two different types of components (*software*[685], respectively hardware, the latter including, besides the digital computer, also *controller*[183]s (*sensor*[659]s, *actuator*[17]s)) performing essentially the same function by cooperating on computing *"that same"* function. (Typically we speak of, i.e., deploy hybridicity when *monitor*[466]ing and *control*[182]ling *reactive system*[579]s — but then hybridicity additionally, to us, means a combination in which the *controller*[183] handles analog matters of continuity, and the *software*[685] plus computer handles discrete matters. Finally, for a conventional analogue *controller*[183] there is usually but one "decision mode". With the software-directed computing system there is now the possibility of multiple discrete + continuous *controller*[183] "regimes".)

347. **Hypothesis:** An assumption made for the sake of argument.

. . . . . $\mathcal{I}$

348. **Icon:** A pictorial representation, an image, a sign whose form (shape, etc.) suggests its meaning. (A graphic symbol on a computer display screen which suggests the purpose of an available *function*[310] or *value*[802] which *designate*[222]s that *entity*[272].)

349. **Iconic:** Adjective form of *icon*[348].

350. **Identification:** The pointing out of a relation, an association, between an *identifier*[351] and that "thing", that *phenomenon*[524], it *designate*[222]s, i.e., it stands for or identifies.

351. **Identifier:** A name. (Usually represented by a string of alphanumeric characters, sometimes with properly infixed "-"s or "_"s.)

352. **Imperative:** Expressive of a command [214]. (We take imperative to more specifically be a reflection of *do this, then do that.* That is, of the use of a *state*[705]-based programming approach, i.e., of the use of an *imperative programming language*[354]. See also *indicative*[362], *optative*[499], and *putative*[566].)

353. **Imperative programming:** Programming, *imperative*[352]ly, "with" references to *storage*[715] *location*[431]s and the updates of those, i.e., of *state*[705]s. (Imperative programming seems to be the classical, first way of programming digital computers.)

354. **Imperative programming language:** A programming language which, significantly, offers language constructs for the creation and manipulation of variables, i.e., *storage*[715]s and their *location*[431]s. (Typical imperative programming languages were, in "ye olde days", `Fortran, Cobol, Algol 60, PL/I, Pascal, C,` etc. [167, 165, 12, 166, 12, 143]. Today programming languages like `C++, Java, C#,` etc. [216, 208, 125] additionally offer *module*[464] cum *object*[487] "features".)

355. **Implementation:** By an implementation we understand a computer program that is made suitable for *compilation*[124] or *interpretation*[397] by a *machine*[436]. (See next entry: *implementation relation*[356].)

356. **Implementation relation:** By an *implementation*[355] relation we understand a logical relation of *correctness*[186] between a *software design specification*[689] and an *implementation*[355] (i.e., a computer program made suitable for *compilation*[124] or *interpretation*[397] by a *machine*[436]).

357. **Incarnation:** A particular instance of a value, usually a state. (We shall here use the term incarnation to designate any one activation on an *activation stack*[13] — where such an incarnation, i.e., activation, represents a program *block*[89] or *function*[310] (or procedure, or *subroutine*[723]) *invocation*[402].)

358. **Incomplete:** We say that a *proof system*[557] is incomplete if not all true sentences are provable.

359. **Incompleteness:** Noun form of the *incomplete*[358] adjective.

360. **Inconsistent:** A set of *axiom*[75]s is said to be inconsistent if, by means of these, and some *deduction rule*[206]s, one can *prove* a property and its negation.

361. **Indefinite:** Not definite, i.e., of a fixed number or a specific property, but it is not known, at the point of uttering the term 'indefinite', what that number or property is. (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

362. **Indicative:** Stating an objective fact. (See also *imperative*[352], *optative*[499] and *putative*[566].)

363. **Induce:** The use of *induction*[364]. (To conclude a general property from special cases.)

364. **Induction:** Inference of a general property from particular instances. (On the basis of several, "similar" cases one may infer a general, say, principle or property. In contrast to *deduction*[205]: from general (e.g., from laws) to specific instances.)

365. **Inductive:** The use of *induction*[364].

366. **In extension:** A concept of logic. In extension is a correlative word that indicates the reference of a term or concept. (When we speak of functions in extension, we shall therefore mean it in the sense of presenting "all details", the "inner workings" of that function. Contrast to *in intension*[378].)

367. **Inert:** A *dynamic*[260] *phenomenon*[524] is said to be inert if it cannot change *value*[802] of its own volition, i.e., by itself, but only through the *interaction*[392] between that *phenomenon*[524] and a change-instigating *environment*[275]. An inert phenomenon only changes value as the result of external stimuli. These stimuli prescribe exactly which new value they are to change to. (Contrast to *active*[14] and *reactive*[578].)

368. **Infer:** Common term for *deduce*[204] or *induce*[363].

369. **Inference rule:** Same as *deduction rule*[206].

370. **Infinite:** As you would think of it: not finite! (Watch out for the four terms: *finite*[288], *infinite*[370], *definite*[209] and *indefinite*[361].)

371. **Informal:** Not formal! (We normally, by an informal specification mean one which may be precise (i.e., unambiguous, and even concise), but which, for example is expressed in natural, yet (domain specific) professional language — i.e., a language which does not have a precise semantics let alone a formal *proof system*[557]. The UML notation is an example of an informal language [181].)

372. **Informatics:** The confluence of (i) *applications*, (ii) *computer science*, (iii) *computing science* [i.e., the art [144, 145, 146] (1968–1973), craft [201] (1981), discipline [82] (1976), logic [123] (1984), practice [124] (1993–2004), and science [114] (1981) of programming], (iv) *software engineering* and (v) *mathematics*.

373. **Information:** The communication or reception of knowledge. (By information we thus mean something which, in contrast to *data*[193], informs us. No computer representation is, let alone any efficiency criteria are, assumed. Data as such does, i.e., bit patterns do, not 'inform' us.)

374. **Information structure:** By an information structure we shall normally understand a composition of more "formally" represented (i.e., structured) *information*[373], for example, in the "believed" form of *table*[739], a *tree*[777], a *graph*[327], etc. (In contrast to *data structure*[199], an information structure does not necessarily have a computer representation, let alone an "efficient" such.)

375. **Informative documentation:** By informative documentation we understand texts which *inform,* but which do not (essentially) describe that which a *development*[228] is to develop. (Informative documentation is balanced by *descriptive* and *analytic*[40] *documentation* to make up the full documentation of a *development*[228].)

376. **Infrastructure:** According to the World Bank: *'Infrastructure' is an umbrella term for many activities referred to as 'social overhead capital' by some development economists, and encompasses activities that share technical and economic features (such as economies of scale and spillovers from users to nonusers).* We shall use the term as follows: Infrastructures are concerned with supporting other systems or activities. Computing systems for infrastructures are thus likely to be distributed and concerned in particular with supporting communication of information, control, people and materials. Issues of (for example) openness, timeliness, security, lack of corruption, and resilience are often important. (Winston Churchill is quoted to have said, during a debate in the House of Commons, in 1946: *... The young Labourite speaker that we have just listened to, clearly wishes to impress upon his constituency the fact that he has gone to Eton and Oxford since he now uses such fashionable terms as 'infra-structures'.*)

377. **Inheritance:** The act of inheriting' a 'property. (The term inheritance, in software engineering, is deployed in connection with a relationship between two pieces (i.e., *module*[464]s) of specification and/or program texts A and B. B may be said to *inherit* some *type*[782], or *variable*[803], or *value*[802] definitions from A.)

378. **In intension:** A concept of logic: In intension is a correlative word that indicates the internal content of a term or concept that constitutes its formal definition. (When we speak of functions in intension, we shall therefore mean it in the sense of presenting only the "input/output" relation of the function. Contrast to *in extension*[366].)

379. **Injection:** A mathematical function, f, that is a one-to-one mapping from *definition set*[211] A to *range set*[577] B. (That is, if for some a in A, $f(a)$ yields a b, then for all $a : A$ all $b : B$ are yielded and there is a unique a for each b, or, which is the same, there is an *inverse function*[401], f^{-1}, such that $f^{-1}(f(a)) = a$ for all $a : A$. See also *bijection*[86] and *surjection*[726].)

380. **Injective function:** A *function*[310] which maps *value*[802]s of its postulated *definition set*[211] into some, but not all, of its postulated *range set*[577] is called injective. (See also *bijective function*[87] and *surjective function*[727] .)

381. **In-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First the tree root is visited and "marked" as having been in-order visited. Then for each subtree a subtree in-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is in-order traversed, then just that tree's root is visited (and that tree has then been in-order traversed) and (the leaf) is "marked" as having been visited. After each subtree visit the root (of the tree of which the subtree is a subtree) is revisited, i.e., again "marked" as having been in-order visited. (Cf. Fig. 13: a left to right in-order traversal of that tree yields the following sequence of "markings": AQCQALXLFLAKUKJKZMZKA. Cf. also Fig. 10).

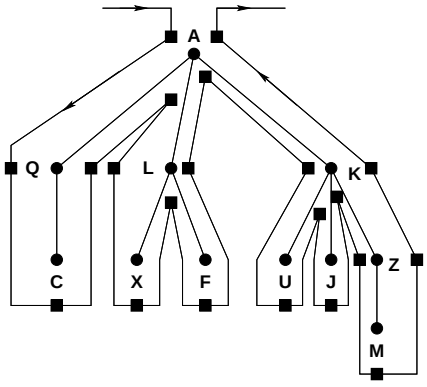

Figure 10: A left-to-right in-order tree traversal

382. **Input:** By input we mean the *communication*[122] of *information*[373] (*data*[193]) from an outside, an *environment*[275], to a *phenomenon*[524] "within" our universe of discourse. (More colloquially, and more generally: Input can be thought of as *value*[802](s) transferred over *channel*[110](s) to, or between *process*[544]es. Cf. *output*[502]. In a narrow sense we talk of input to an *automaton*[71] (i.e., a *finite state automaton*[289] or a *pushdown automaton*[564]) and a *machine*[436] (here in the sense of, for example, a *finite state machine*[290] (or a *pushdown machine*[565])).)

383. **Input alphabet:** The set of *symbol*[728]s *input*[382] to an *automaton*[71] or a *machine*[436] in the sense of, for example, a *finite state machine*[290] or a *pushdown machine*[565].

384. **Instance:** An individual, a thing, an *entity*[272]. (We shall usually think of an 'instance' as a *value*[802].)

385. **Instantiation:** 'To represent (an abstraction) by a concrete *instance*[384]' [214]. (We shall sometimes be using the term 'instantiation' in lieu of a *function invocation*[317] on an *activation stack*[13].)

386. **Installation manual:** A *document*[237] which describes how a *computing system*[151] is to be installed. (A special case of 'installation' is the downloading of *software*[685] onto a *computing system*[151]. See also *training manual*[767] and *user manual*[798].)

387. **Intangible:** Not *tangible*[742].

388. **Integrity:** By a *machine*[436] having integrity we mean that that machine remains unimpaired, i.e., has no faults, errors and failures, and remains so even in the situations where the environment of the machine has faults, errors and failures. (Integrity is a *dependability requirement*[218].)

389. **Intension:** Intension indicates the internal content of a term. (See also *in intension*[378]. The intension of a *concept*[152] is the collection of the properties possessed jointly by all conceivable individuals falling under the concept [179]. The intension determines the *extension*[283] [179].)

390. **Intensional:** Adjective form of *intension*[389].

391. **Interact:** The term interact here addresses the phenomenon of one *behaviour*[79] acting in unison, simultaneously, *concurrent*[159]ly, with another behaviour, including one behaviour influencing another behaviour. (See also *interaction*[392].)

392. **Interaction:** Two-way reciprocal action.

393. **Interface:** Boundary between two disjoint sets of communicating phenomena or concepts. (We shall think of the systems as *behaviour*[79]s or *process*[544]es, the boundary as being *channel*[110]s, and the communications as *input*[382]s and *output*[502]s.)

394. **Interface requirements:** Interface requirements are those *requirements*[605] which can on be expressed using professional, i.e., technical terms from *both* the *domain*[239] and the *machine*[436]. Thus, by interface requirements we understand the expression of expectations as to which software-software, or software-hardware *interface*[393] places (i.e., *channel*[110]s), *input*[382]s and *output*[502]s (including the *semiotics*[658] of these input/outputs) there shall be in some contemplated *computing system*[151]. (Interface requirements can often, usefully, be classified in terms of *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *computational data+control*

requirements[146], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448] and *machine-machine dialogue requirements*[437]. Interface requirements constitute one requirements *facet*[285]. Other requirements facets are: *business process reengineering*[101], *domain requirements*[258] and *machine requirements*[438].)

395. **Interface requirements facet:** See *interface requirements*[394] for a list of facets: *shared data initialisation*[670], *shared data refreshment*[672], *computational data+control*[146], *man-machine dialogue*[446], *man-machine physiological*[448] and *machine-machine dialogue*[437] *requirements*[605].

396. **Interpret:** See next: *interpretation*[397].

397. **Interpretation:** The three terms *elaboration*, *evaluation*[280] and *interpretation*[397] essentially cover the same idea: that of obtaining the meaning of a syntactical item in some *configuration*[160], or as a function from configurations to *value*[802]s. Given that configuration typically consists of *static*[708] *environment*[275]s and *dynamic*[260] *state*[705]s (or *storage*[715]s), we use the term interpretation in the more narrow sense of designating, or yielding functions from syntactical items to functions from configurations to states.

398. **Interpreter:** An interpreter is an *agent*[24], a *machine*[436], which performs *interpretation*[397]s.

399. **Intrinsics:** By the intrinsics of a *domain*[239] we shall understand those phenomena and concepts of a domain which are basic to any of the other facets, with such a domain intrinsics initially covering at least one specific, hence named, *stakeholder*[703] view. (Intrinsics is thus one of several *domain facet*[250]s. Others include: *business process*[99]es, *support technology*[725], *rules and regulations*[640], *scripts*[651], *management and organisation*[445], and *human behaviour*[345].)

400. **Invariant:** By an invariant we mean a property that holds of a *phenomenon*[524] or a *concept*[152], both before and after any *action*[12] involving that phenomenon or a concept. (A case in point is usually an *information*[373] or a *data structure*[199]: Assume an action, say a repeated one (e.g., a while loop). We say that the action (i.e., the while loop) preserves an invariant, i.e., usually a *proposition*[560], if the proposition holds true of the *state*[705] before and the state after any *interpretation*[397] of the while loop. Invariance is here seen separate from the *well-formedness*[812] of an *information*[373] or a *data structure*[199]. We refer to the explication of *well-formedness*[812]!)

401. **Inverse function:** See *injection*[379].

402. **Invocation:** See *function invocation*[317].

403. **Isomorphic:** One to one. (See *isomorphism*[404].)

404. **Isomorphism:** If a *homomorphism*[343] ϕ is a *bijective function*[87] then ϕ is an isomorphism. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276] and *monomorphism*[467].)

. . . . . $\mathcal{J}$

405. **J:** The **J** operator (**J** for **J**ump) was introduced (before 1965) by Peter Landin as a *functional*[312] used to explain the creation and use of program *closure*[117]s, and these again are used to model the *denotation*[213] of *label*[410]s. (We refer to [151, 153, 152, 150, 74]. Cf. `www.dcs.qmw.ac.uk/~peterl/danvy/`.)

. . . . . $\mathcal{K}$

406. **Keyword:** A significant word from a title or document.

407. **Knowledge:** What is, or what can be known. The body of truth, information, and principles acquired by mankind [214]. (See *epistemology*[277] and *ontology*[492]. *A priori knowledge:* Knowledge that is independent of all particular experiences. *A posteriori knowledge:* Knowledge, which derives from experience alone.)

408. **Knowledge engineering:** The representation and modelling of knowledge. (The construction of ontological and epistemological knowledge and its manipulation. Involves such subdisciplines as *modal logic*[459]s (promise and commitment, knowledge and belief), *speech act* theories, *agent*[24] theories, etc. Knowledge engineering usually is concerned with the knowledge that one agent may have about another agent.)

409. **KWIC:** Abbreviation for keyword-in-context (A classical software application.)

. . . . . $\mathcal{L}$

410. **Label:** Same as named *program point*[548].

411. **Lambda-application:** Within the confines of the *Lambda-calculus*[412], *Lambda-application*[411] is the same as *function application*[315]. (Subject, however, to simple *term-rewriting*[753] using (say just) *Alpha-renaming*[35] and *Beta-reduction*[84].)

412. **Lambda-calculus:** A *calculus*[104] for expressing and "manipulating" functions. The Lambda-calculus (λ-calculus) is a de facto "standard" for "what is computable". See *Lambda-expression*[414]s. As a *calculus*[104] it prescribes a language, the language of *Lambda-expression*[414]s, a set of *conversion*[184] rules — these apply to *Lambda-expression*[414]s and result in *Lambda-expression*[414]s. They "mimic" *function definition*[316] and *function application*[315]. The seminal texts on the Lambda-calculi are [66, 13, 14, 15].

413. **Lambda-combination:** See *Lambda-application*[411].

414. **Lambda-expression:** The language of the "pure" (i.e., simple, but fully powerful) *Lambda-calculus*[412] has three kinds of Lambda-expressions: *Lambda-variable*[416]s, *Lambda-function*[415]s and *Lambda-application*[411]s.

415. **Lambda-function:** By a Lambda-function we understand a *Lambda-expression*[414] of the form $\lambda x \bullet e$, where x is a binding variable and e is a Lambda-expression. (It is usually the case that e contains *free*[305] occurrences of x — these being bound by the binding variable in $\lambda x \bullet e$.)

416. **Lambda-variable:** The x in the *Lambda-function*[415] expression $\lambda x \bullet e$: both the formal parameter, the first x you see in $\lambda x \bullet e$, and all the *free*[305] occurrences of x in the *block*[89] (i.e., body) expression e.

417. **Language:** By a language we shall understand a possibly infinite set of *sentence*[660]s which follow some *syntax*[733], express some *semantics*[655] and are uttered, or written down, due to some *pragmatics*[534].

418. **Law:** A law is a rule of conduct prescribed as binding or enforced by a controlling authority. (We shall take the term law in the specific sense of law of Nature (cf., Ampére's Law, Boyle's Law, the conservation laws (of mass-energy, electric charge, linear and angular momentum), Newton's Laws, Ohm's Law, etc.), and laws of Mathematics (cf. "law of the excluded middle" (as in logic: a proposition must either be true, or false, not both, and not none)).)

419. **Leaf:** A leaf is a *node*[479] in a *tree*[777] for which there are no sub*tree*[777]s of that node. (Thus a leaf is a concept of *tree*[777]s. Cf. Fig. 13 on page 233.)

420. **Lemma:** An auxiliary *proposition*[560] used in the demonstration of another proposition. (Instead of proposition we could use the term *theorem*[756].)

421. **Lexical analysis:** The analysis of a *sentence*[660] into its constituent *word*[814]s. (Sentences also are usually "decorated" with such signs as for example punctuation marks (, . : ;), delimiters (( ) [ ], etc.), and other symbols (? !, etc.). Lexical analysis therefore is a process which serves to recognise which character sequences are words and which are not (i.e., which are delimiters, etc.).)

422. **Lexicographic:** The principles and practices of establishing, maintaining and using a dictionary. (We shall, in software engineering, mostly be using the term 'lexicographic' in connection with compilers and, more rarely, database schemas — although, as the definition implies, it is of relevance in any context where a computing system builds, maintains and uses a dictionary.)

423. **Lexicographical order:** The order, i.e., sequence, in which entries of a dictionary appear. (More specifically, the lexicographical ordering of entries in a *compiler dictionary*[126] is, for a *block-structured programming language*[90], determined by the nesting

structure of *block*[89]s. The dictionary itself, generally "mimics" the nesting structure of the language.)

424. **License:** A license is a *script*[651] specifically expressing a permission to act; is freedom of action; is a permission granted by competent authority to engage in a business or occupation or in an activity otherwise unlawful; a document, plate, or tag evidencing a license granted; a grant by the holder of a copyright or patent to another of any of the rights embodied in the copyright or patent short of an assignment of all rights. Licenses appear more to have morally than legally binding poser.

425. **Link:** A link is the same as a *pointer*[528], an *address*[22] or a *reference*[587]: something which refers to, i.e., designates something (typically something else).

426. **Lifted function:** A lifted function, say of type $A \to B \to C$, has been created from a function of type $B \to C$ by 'lifting' it, i.e., by abstracting it in a variable, say a of type A. (Assume $\lambda b : B \cdot \mathcal{E}(b)$ to be a function of type $B \to C$. Now $\lambda a : A \cdot \lambda b : B \cdot \mathcal{E}(b)$ is a lifted version of $\lambda b : B \cdot \mathcal{E}(b)$. An example is **and**: $\lambda b_1, b_2 : \mathbf{Bool} \cdot b_1 \wedge b_2$, Boolean conjunction. We lift **and** to be a function, $\wedge_T$, over time: $\lambda t : T \cdot b_1(t) \wedge b_2(t)$, where the variables b_1, b_2 typically could be (e.g., assignable) variables whose values change over time.)

427. **Linguistics:** The study and knowledge of the *syntax*[733], *semantics*[655] and *pragmatics*[534] of *language*[417](s).

428. **List:** A list is an ordered sequence of zero, one or more not necessarily distinct entities.

429. **Literal:** A term whose use in software engineering, i.e., programming, shall mean: an identifier which denotes a constant, or is a keyword. (Usually that identifier is emphasised. Examples of RSL literals are: **Bool, true, false, chaos, if, then, else, end, let, in,** and the numerals $0, 1, 2., ..., 1234.5678$, etc.)

430. **Live Sequence Chart:** The Live Sequence Chart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [121].)

431. **Location:** By a location is meant an area of *storage*[715].

432. **Logic:** The principles and criteria of validity of inference and deduction, that is, the mathematics of the formal principles of reasoning. (We refer to Vol. 1, Chap. 9 for our survey treatment of mathematical logic.)

433. **Logic programming:** Logic programming is programming based on an interpreter which either performs deductions or inductions, or both. (In logic programming the chief values are those of the Booleans, and the chief forms of expressions are those of propositions and predicates.)

434. **Logic programming language:** By a *logic programming*[433] language is meant a language which allows one to express, to prescribe, *logic programming*[433]. (The classical logic programming language is `Prolog` [161, 133].)

435. **Loose specification:** By a loose specification is understood a specification which either *underspecifies* a problem, or specifies this problem *nondeterministically*.

. . . . . $\mathcal{M}$

436. **Machine:** By the machine we understand the *hardware*[331] plus *software*[685] that implements some *requirements*[605], i.e., a *computing system*[151]. (This definition follows that of M.A. Jackson [139].)

437. **Machine-Machine dialogue requirements:**By machine-machine dialogue requirements we understand the *syntax*[733] (incl. sequential structure), and *semantics*[655] (i.e., meaning) of the communications (i.e., messages) transferred in either direction over the automated interface between *machine*[436]s (including supporting technologies). (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], and *man-machine physiological requirements*[448].)

438. **Machine requirements:** Machine requirements are those *requirements*[605] which, in principle, can be expressed without using professional (i.e., technical) terms from the *domain*[239] (for which these requirements are established). Thus, by *machine*[436] *requirements*[605] we understand *requirements*[605] put specifically to, i.e., expected specifically from, the *machine*[436]. (We normally analyse machine requirements into *performance requirements*[521], *dependability requirements*[218], *maintenance requirements*[443], *platform requirements*[527] and *documentation requirements*[238].)

439. **Machine service:** The service delivered by a machine is its *behaviour*[79] as it is perceptible by its user(s), where a user is a human, another machine, or a(nother) system which *interact*[391]s with it [197].

440. **Macro:** Macros have the same syntax as procedures, that is, a pair of a *signature*[680] (i.e., a macro name followed by a formal argument list of distinct identifiers (i.e., the *formal parameter*[302]s)) and a macro body, a text. Syntactically we can distinguish between macro definitions and macro *invocation*[402]s. Semantically, invocations, in some text, of the macro name and an *actual argument*[16] list are then to be thought of as an expansion of that part of the text with the macro (definition) body and such that formal parameters are replaced (*macro substitution*[441]) by actual arguments. Semantically a macro is different from a *procedure*[543] in that a macro expansion takes place in a *context*[172], i.e., an *environment*[275], where *free*[305] identifiers of the macro body are replaced by their value as defined at the place of the occurrence of the macro invocation. Whereas, for a procedure, the free identifiers of a procedure body are bound to their value at the point where the procedure was defined. (Thus the

difference between a macro and a procedure is the difference between *evaluation*[280] in a calling, versus in a defining environment.)

441. **Macro substitution:** See under *macro*[440]s.

442. **Maintenance:** By maintenance we shall here, for software, mean change to *software*[685], i.e., its various *document*[237]s, due to needs for (i) adapting that software to new *platform*[526]s, (ii) correcting that software due to observed software errors, (iii) improving certain performance properties of the *machine*[436] of which the software is part, or (iv) avoiding potential problems with that machine. (We refer to subcategories of maintenance: *adaptive maintenance*[21], *corrective maintenance*[187], *perfective maintenance*[519] and *preventive maintenance*[541].)

443. **Maintenance requirements:** By *maintenance*[442] *requirements*[605] we understand requirements which express expectations on how the *machine*[436] being desired (i.e., required) is expected to be maintained. (We also refer to *adaptive maintenance*[21], *corrective maintenance*[187], *perfective maintenance*[519] and *preventive maintenance*[541].)

444. **Management:** Management is about resources: their acquisition, scheduling (over time), allocation (over locations), deployment (in performing actions) and disposal ("retirement"). (We distinguish between board-directed, strategic, tactical and operational actions: *board-directed* actions target mainly financial resources: obtaining new funds through conversion of goodwill into financial resources, acquiring and selling "competing" or "supplementary" business units; *strategic* actions convert financial resources into production, service supplies and resources and vice-versa — and in this these actions schedule availability of such resources; *tactical* actions mainly allocate resources; and *operational* actions order, monitor and control the deployment of resources in the performance of actions.)

445. **Management and organisation:** The composite term management and organisation applies in connection with *management*[444] as outlined just above and with *organisation*[500]. The term then emphasises the relations between the organisation and management of an enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *rules and regulations*[640] and *human behaviour*[345].)

446. **Man-machine dialogue:** By man-machinedialogues we understand actual instantiations of *user*[796] interactions with *machine*[436]s, and machine interactions with users: what input the users provide, what output the machine initiates, the interdependencies of these inputs/outputs, their temporal and spatial constraints, including response times, input/output media (locations), etc. (

447. **Man-machine dialogue requirements:** By man-machine dialogue requirements we understand those *interface requirements*[394] which express expectations on, i.e.,

mandates the *protocol*[561] according to which *user*[796]s are to interact with the *machine*[436], and the machine with the users. (See *man-machine dialogue*[446]. For other *interface requirements*[394] see *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine physiological requirements*[448] and *machine-machine dialogue requirements*[437].))

448. **Man-machine physiological requirements:** By man-machine physiological requirements we understand those *interface requirements*[394] which express expectations on, i.e., mandates, the form and appearance of ways in which the *man-machine dialogue*[446] utilises such physiological devices as visual display screens, keyboards, "mouses" (and other tactile instruments), audio microphones and loudspeakers, television cameras, etc. (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447] and *machine-machine dialogue requirements*[437].)

449. **Map:** A map is like a *function*[310], but is here thought of as an *enumerable*[273] set of pairs of argument/result values. (Thus the *definition set*[211] of a map is usually decidable, i.e., whether an entity is a member of a definition set of a map or not can usually be decided.)

450. **Mechanical semantics:** By a mechanical semantics we understand the same as an *operational semantics*[496] (which is again basically the same as a *computational semantics*[147]), i.e., a semantics of a language specified using concrete constructs (like stacks, program pointers, etc.), and otherwise as defined in *operational semantics*[496] and *computational semantics*[147].

451. **Mereology:** The theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole. (Mereology is often considered a branch of *ontology*[492]. Leading investigators of mereology were Franz Brentano, Edmund Husserl, Stanislaw Lesniewski [209, 164, 175, 212, 213, 217] and Leonard and Goodman [156].)

452. **Meta-IV:** `Meta-IV` stands for the fourth metalanguage (for programing language definition conceived at the IBM Vienna Laboratory in the 1960s and 1970s). (`Meta-IV` is pronounced meta-four.)

453. **Metalanguage:** By a metalanguage is understood a *language*[417] which is used to explain another language, either its *syntax*[733], or its *semantics*[655], or its *pragmatics*[534], or two or all of these! (One cannot explain any language using itself. That would lead to any interpretation of what is explained being a valid solution, in other words: Nonsense. `RSL` thus cannot be used to explain `RSL`. Typically formal specification languages are metalanguages: being used to explain, for example, the semantics of ordinary programming languages.)

454. **Metalinguistic:** We say that a language is used in a metalinguistic manner when it is being deployed to explain some other language. (And we also say that when we examine a language, like we could, for example, examine RSL, and when we use a subset of RSL to make that analysis, then that subset of RSL is used metalinguistically (wrt. all of RSL).)

455. **Metaphysics:** We quote from: http://mally.stanford.edu/: "Whereas physics is the attempt to discover the laws that govern fundamental concrete objects, metaphysics is the attempt to discover the laws that systematize the fundamental abstract objects presupposed by physical science, such as natural numbers, real numbers, functions, sets and properties, physically possible objects and events, to name just a few. The goal of metaphysics, therefore, is to develop a formal ontology, i.e., a formally precise systematization of these abstract objects. Such a theory will be compatible with the world view of natural science if the abstract objects postulated by the theory are conceived as patterns of the natural world." (Metaphysics may, to other scientists and philosophers, mean more or other, but for software engineering the characterisation just given suffices.)

456. **Method:** By a method we shall here understand a set of *principle*[542]s for selecting and using a number of *technique*[745]s and *tool*[763]s in order to construct some *artefact*[55]. (This is our leading definition — one that sets out our methodological quest: to identify, enumerate and explain the principles, the techniques and, in cases, the tools — notably where the latter are specification and programming languages. (Yes, languages are tools.))

457. **Methodology:** By methodology we understand the study and knowledge of *method*[456]s, one, but usually two or more. (In some dialects of English, methodology is confused with method.)

458. **Mixed computation:** By a mixed computation we understand the same as by a *partial evaluation*[516]. (The term mixed computation was used notably by Andrei Petrovich Ershov [99, 104, 105, 98, 100, 101, 102, 103], in my mind the "father" of Russian computing science.)

459. **Modal logic:** A modal is an expression (like "necessarily" or "possibly") that is used to qualify the truth of a judgment. Modal logic is, strictly speaking, the study of the deductive behavior of the expressions "it is necessary that" and "it is possible that". (The term "modal logic" may be used more broadly for a family of related systems. These include logics for belief, for tense and other temporal expressions, for the deontic (moral) expressions such as "it is obligatory that", "it is permitted that" and many others. An understanding of modal logic is particularly valuable in the formal analysis of philosophical argument, where expressions from the modal family are both common and confusing. Modal logic also has important applications in computer science [235].)

460. **Model:** A model is the mathematical meaning of a description (of a domain), or a prescription (of requirements), or a specification (of software), i.e., is the meaning of a specification of some universe of discourse. (The meaning can be understood either as a mathematical function, as for a *denotational semantics*[215] meaning, or an *algebra*[26] as for an *algebraic semantics*[27] or a *denotational semantics*[215] meaning, etc. The essence is that the model is some mathematical structure.)

461. **Model-oriented:** A specification (description, prescription) is said to be model-oriented if the specification (etc.) *denote*[216]s a *model*[460]. (Contrast to *property-oriented*[559].)

462. **Model-oriented type:** A type is said to be model-oriented if its specification *designate*[222]s a *model*[460]. (Contrast to *property-oriented*[559] *type*[782].)

463. **Modularisation:** The act of structuring a text using *module*[464]s.

464. **Module:** By a module we shall understand a clearly delineated text which denotes either a single complex quantity, as does, usually, an *object*[487], or a possibly empty, possibly infinite set of *model*[460]s of objects. (The RSL module concept is manifested in the use of one or more of the RSL *class*[114] (**class ... end**), *object*[487] (**object** identifier **class ... end**, etc.), and *scheme*[648] (**scheme** identifier **class ... end**), etc., constructs. We refer to [73, 72, 23] and to [187, 186] for original, early papers on modules.)

465. **Module design:** By module design we shall understand the *design*[221] of (one or more) *module*[464]s.

466. **Monitor:** Syntactically a monitor is "a programming language construct which encapsulates variables, access procedures and initialisation code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections." Semantically "a monitor may have a queue of processes which are waiting to access it" [108].

467. **Monomorphism:** If a *homomorphism*[343] ϕ is an *injective function*[380] then ϕ is an isomorphism. (See also *automorphism*[72], *endomorphism*[268], *epimorphism*[276], and *monomorphism*[467].)

468. **Monotonic:** A function, $f : A \to B$, is monotonic, if for all a, a' in the definition set A of f, and some ordering relations, $\sqsubseteq$, on a and B, we have that if $a \sqsubseteq a'$ then $f(a) \sqsubseteq f(a')$.

469. **Mood:** A conscious state of mind, as here, of a specification. (We can thus express an *indicative*[362] mood, an *optative*[499] mood, a *putative*[566] mood or an *imperative*[352] mood. Our use of these various forms of moods is due to Michael Jackson [139].)

470. **Morphism:** Same as *homomorphism*[343].

471. **Morphology:** (i) A study and description of word formation (as inflection, derivation, and compounding) in language; (ii) the system of word-forming elements and processes in a language; (iii) a study of structure or form [214].

472. **Multi-dimensional:** A composite (i.e., a non*atomic*[63]) *entity*[272] is a multi-dimensional *entity*[272] if some relations between properly contained (i.e., constituent) subentities (cf. *subentity*[721]) can only be described by both forward and backward references, and/or with recursive references. (This is in contrast to *one-dimensional*[491] entities.)

473. **Multimedia:** The use of various forms of input/output media in the man-machine interface: Text, two-dimensional graphics, voice (audio), video, and tactile instruments (like "mouse").

. . . . . $\mathcal{N}$

474. **Name:** A name is syntactically (generally an expression, but usually it is) a simple alphanumeric identifier. Semantically a name denotes (i.e., designates) "something". Pragmatically a name is used to uniquely identify that "something". (Shakespeare: Romeo: "What's in a name?" Juliet to Romeo: "That which we call a rose by any other name would smell as sweet.")

475. **Naming:** The action of allocating a unique name to a value.

476. **Narrative:** By a narrative we shall understand a document text which, in precise, unambiguous language, introduces and describes (prescribes, specifies) all relevant properties of entities, functions, events and behaviours, of a set of phenomena and concepts, in such a way that two or more readers will basically obtain the same idea as to what is being described (prescribed, specified). (More commonly: Something that is narrated, a story.)

477. **Natural language:** By a natural language we shall understand a language like Arabic, Chinese, English, French, Russian, Spanish, etc. — one that is spoken today, 2005, by people, has a body of literature, etc. (In contrast to natural languages we have (i) professional languages, like the languages of medical doctors, or lawyers, or skilled craftsmen like carpenters, etc.; and we have (ii) formal languages like software specification languages, programming languages, and the languages of first-order predicate logics, etc.)

478. **Network:** By a network we shall understand the same as a directed, but not necessarily *acyclic graph*[19]. (Our only use of it here is in connection with network *databases*.)

479. **Node:** A point in some *graph*[327] or *tree*[777].

480. **Nondeterminate:** Same as *nondeterministic*[481].

481. **Nondeterministic:** A property of a specification: May, on purpose, i.e., deliberately have more than one meaning. (A specification which is ambiguous also has more than one meaning, but its ambiguity is of overriding concern: It is not 'nondeterministic' (and certainly not 'deterministic'!).)

482. **Nondeterminism:** A *nondeterministic*[481] specification models nondeterminism.

483. **Nonstrict:** Nonstrictness is a property associated with functions. A function is nonstrict, in certain or all arguments, if, for undefined values of these it may still yield a defined value. (See also *strict function*[717]s.)

484. **Nonterminal:** The concept of a nonterminal (together with the concept of a *terminal*[750]) is a concept associated with the *rule of grammar*[639]s. (See that term: *rule of grammar*[639] for a full explanation.)

485. **Notation:** By a notation we shall usually understand a reasonably precisely delineated language. (Some notations are textual, as are programming notations or specification languages; some are diagrammatic, as are, for example, *Petri net*[522]s, *Statechart*[706]s, *Live Sequence Chart*[430]s, etc.)

486. **Noun:** Something, a name, that refers to an *entity*[272], a quality, a *state*[705], an *action*[12], or a *concept*[152]. Something that may serve as the subject of a *verb*[806]. (But beware: In English many nouns can be "verbed", and many verbs can be "nouned"!)

. . . . . *O*

487. **Object:** An instance of the *data structure*[199] and *behaviour*[79] defined by the object's *class*[114]. Each object has its own *value*[802]s for the instance *variable*[803]s of its class and can respond to the *function*[310]s defined by its class. (Various *specification language*[699]s, `object Z` [61, 95, 96], `RSL`, etc., each have their own, further refined, meaning for the term 'object', and so do *object-oriented*[488] *programming language*[551] (viz., `C++` [216], `Java` [10, 113, 158, 225, 6, 208], `C#` [192, 174, 173, 125] and so on).)

488. **Object-oriented:** We say that a program is *object-oriented*[488] if its main structure is determined by a *modularisation*[463] into a *class*[114], that is, a cluster of *type*[782]s, *variable*[803]s and *procedure*[543]s, each such set acting as a separate *abstract data type*[4]. Similarly we say that a *programming language*[551] is object-oriented if it specifically offers language constructs to express the appropriate *modularisation*[463]. (Object-orientedness became a mantra of the 1990s: Everything had to be object-oriented. And many programming problems are indeed well served by being structured around some object-oriented notion. The first *object-oriented*[488] *programming language*[551] was `Simula 67` [23].)

489. **Observer:** By an observer we mean basically the same as an *observer function*[490].

490. **Observer function:** An observer function is a *function*[310] which when "applied" to an *entity*[272] (a *phenomenon*[524] or a *concept*[152]) yields subentities or attributes of that entity (without "destroying" that entity). (Thus we do not make a distinction between functions that observe subentities (cf. *subentity*[721]) and functions that observe *attribute*[69]s. You may wish to make distinctions between the two kinds of observer function. You can do so by some simple *naming*[475] convention: assign names the prefix obs_ when you mean to observe subentities, and attr_ when you mean to observe attributes. Vol. 3 Chap. 5 introduces these concepts.)

491. **One-dimensional:** A composite *entity*[272] is a one-dimensional *entity*[272] if all relations between properly contained (i.e., constituent) subentities can be described by either no references to other subentities, or only by backward or only by forward references. (This is in contrast to *multi-dimensional*[472] entities. Thus arrays of arbitrary order (vectors, matrices, tensors) are usually one-dimensional.)

492. **Ontology:** In philosophy: A systematic account of Existence. To us: An explicit formal specification of how to represent the phenomena, concepts and other entities that are assumed to exist in some area of interest (some universe of discourse) and the relationships that hold among them. (Further clarification: An ontology is a catalogue of *concept*[152]s and their relationships — including properties as relationships to other concepts.)

493. **Operation:** By an operation we shall mean a *function*[310], or an *action*[12] (i.e., the effect of function *invocation*[402]). (The context determines which of these two strongly related meanings are being referred to.)

494. **Operational:** We say that a *specification*[698] (a *description*[220], a *prescription*[540]), say of a *function*[310], is operational if what it explains is explained in terms of how that thing, **how** that phenomenon, or concept, operates (rather than by **what** it achieves). (Usually operational definitions are *model oriented*[461] (in contrast to *property oriented*[559]).)

495. **Operational abstraction:** Although a definition (a *specification*[698], a *description*[220], or a *prescription*[540]) may be said, or claimed, to be *operational*[494], it may still provide *abstraction*[3] in that the *model-oriented*[461] concepts of the definition are not themselves directly representable or performable by humans or computers. (This is in contrast to *denotational*[214] *abstraction*[3]s or *algebra*[26]ic (or *axiom*[75]atic) *abstraction*[3]s.)

496. **Operational semantics:** A *definition*[210] of a *language*[417] *semantics*[655] that is *operational*[494]. (See also *structural operational semantics*[720].)

497. **Operation reification:** To speak of *operation*[493] *reification*[597] one must first be able to refer to an abstract, usually *property-oriented*[559], specification of the operation.

Then, by operation *reification*[597] we mean a *specification*[698] which indicates how the operation might be (possibly efficiently) implemented. (Cf. *data reification*[198] and *operation transformation*[498].)

498. **Operation transformation:** To speak of *operation*[493] *reification*[597] one must first be able to refer to an abstract, usually *property-oriented*[559], specification of the operation. Then, by operation *transformation*[771] we mean a *specification*[698] which is, somehow, *calculate*[102]d from the abstract specification. (Three nice books on such calculi are: [177, 22, 11].)

499. **Optative:** Expressive of wish or desire. (See also *imperative*[352], *indicative*[362], and *putative*[566].)

500. **Organisation:** Organisation is about the "grand scale", executive and strategic national, continental or global (world wide) (i) allocation of major resource (e.g., business) units, whether in a hierarchical, in a matrix, or in some other organigram-specified structure, (ii) as well as the clearly defined relations (which information, decisions and actions are transferred) between these units, and (iii) organisational dynamics.

501. **Organisation and management:** The composite term organisation and management applies in connection with *organisation*[500] as outlined just above and with *management*[444]s (cf. Item 444 on page 195). The term then emphasises the relations between the organisation and management of an enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *rules and regulations*[640] and *human behaviour*[345].)

502. **Output:** By output we mean the *communication*[122] of *information*[373] (*data*[193]) to an outside, an *environment*[275], from a *phenomenon*[524] "within" our universe of discourse. (More colloquially, and more generally: output can be thought of as *value*[802](s) transferred over *channel*[110](s) from, or between, *process*[544]es. Cf. *input*[382]. In a narrow sense we talk of output from a *machine*[436] (e.g., a *finite state machine*[290] or a *pushdown machine*[565]).)

503. **Output alphabet:** The set of *symbol*[728]s *output*[502] from a *machine*[436] in the sense of, for example, a *finite state machine*[290] or a *pushdown machine*[565].

504. **Overloaded:** The concept of 'overloaded' is a concept related to *function*[310] *symbol*[728]s, i.e., *function*[310] *name*[474]s. A function name is said to be overloaded if there exists two or more distinct *signature*[680]s for that function name. (Typically overloaded function symbols are '+', which applies, possibly, in some notation, to addition of integers, addition of reals, etc., and '=', which applies, possibly, in some notation, to comparison of any pair of *value*[802]s of the same *type*[782].)

. . . . . $\mathcal{P}$

505. **Paradigm:** A philosophical and theoretical framework of a scientific school or discipline within which theories, laws and generalizations and the experiments performed in support of them are formulated; a philosophical or theoretical framework of any kind. (Software engineering is full of paradigms: Object-orientedness is one.)

506. **Paradox:** A statement that is seemingly contradictory or opposed to common sense and yet is perhaps true. An apparently sound argument leading to a contradiction. (Some famous examples are Russell's Paradox[16] and the Liar Paradox.[17] Most paradoxes stem from some kind of self-reference.)

507. **Parallel programming language:** A *programming language*[551] whose major kinds of concepts are *process*[544]es, process *composition*[140] [putting processes in parallel and *nondeterministic*[481] {internal or external} choice of process *elaboration*[264]], and synchronisation and communication between processes. (A main example of a practical parallel programming language is `occam` [135], and of a specificational 'programming' language is `CSP` [130, 203, 207]. Most recent *imperative programming language*[354]s (Java, C#, etc.) provide for programming constructs (e.g., threads) that somehow mimic parallel programming.)

508. **Parameter:** Same as *formal parameter*[302].

509. **Parametric polymorphism:** See the parenthesised part of the *polymorphic*[529] entry.

510. **Parameterised:** We say that a *definition*[210], of a *class*[114] (or of a *function*[310]) is parameterised if an *instantiation*[385] of an *object*[487] of the class (respectively an *invocation*[402] of the function) allows an *actual argument*[16] to be substituted (cf. *substitution*[722]) into the class definition (function body) for every occurrence of the [formal] *parameter*[508].

511. **Parser:** A parser is an *algorithm*[31], say embodied as a *software*[685] *program*[545], which accepts text strings, and, if the text string is generated by a suitable *grammar*[325], then it will yield a *parse tree*[512] of that string. (See *generator*[322].)

512. **Parse tree:** To speak of a parse tree we assume the presence of a string of *terminal*[750]s and *nonterminal*[484]s, and of a *grammar*[325]. A parse tree is a *tree*[777] such that each subtree (of a *root*[632] and its immediate descendants, whether *terminal*[750]s or *nonterminal*[484]s) corresponds to a *rule*[638] of the grammar, and hence such that the *frontier*[307] of the tree is the given string.

513. **Parsing:** The act of attempting to construct a *parse tree*[512] from a *grammar*[325] and a text string.

[16] If R is the set of all sets which do not contain themselves, does R contain itself? If it does then it doesn't and vice versa.

[17] "This sentence is false" or "I am lying".

514. **Part:** To speak of parts we must be able to speak of "parts and wholes". That is: We assume some *mereology*[451], i.e., a theory of parthood relations: of the relations of part to whole and the relations of part to part within a whole.

515. **Partial algebra:** A partial *algebra* is an algebra whose functions are not defined for all combinations of arguments over the carrier.

516. **Partial evaluation:** To speak of partial evaluation we must first speak of *evaluation*[280]. Normally evaluation is a *process*[544], as well as the result of that process, whereby an *expression*[282] in some language is evaluated in some *context*[172] which binds every *free identifier*[305] of the expression to some *value*[802]. A partial evaluation is an evaluation in whose context not all free identifiers are bound to (hence, defined) values. The result of a partial evaluation is therefore a symbolic evaluation, one in which the resulting value is expressed in terms of actual values and the undefined free identifiers. (We refer to [51, 142].)

517. **Path:** The concept of paths is usually associated with *graph*[327]s and *tree*[777]s (i.e., networks). A path is then a sequence of one or more graph edges or tree branches such that two consecutive edges (branches) share a node of the graph (or [root] of a tree). (We shall also use the term *route*[635] synonymously with paths.)

518. **Pattern:** We shall take a pattern, p, (as in RSL) to mean an expression with identifiers, a, and constants, k, as follows. Basis clauses: Any identifier a is a pattern, and any constant, k, is a pattern. Inductive clause: If $p_1, p_2, \ldots, p_m$ are patterns, then so are $(p_1, p_2, \ldots, p_m)$, $< p_1, p_2, \ldots, p_m >$, $\{p_1, p_2, \ldots, p_m\}$, $[p_{d_1} \mapsto p_{r_1}, p_{d_2} \mapsto p_{r_2}, \ldots, p_{d_m} \mapsto p_{r_m}]$, and so are: $< p > \hat{} a$, $a \hat{} < p >$, $\{p\} \cup a$, and $[p_{d_1} \mapsto p_{r_1}] \cup a$. (The idea is that a pattern, p, is "held up against" a value, v, "of the same kind" and then we attempt to "match" the pattern, p, with the value, v, and if a matching can be made, then the free identifiers of p are bound to respective component values of v.)

519. **Perfective maintenance:** By perfective maintenance we mean an update, as here, of software, to achieve a more desirable use of resources: time, storage space, equipment. (We also refer to *adaptive maintenance*[21], *corrective maintenance*[187] and *preventive maintenance*[541].)

520. **Performance:** By performance we, here, in the context of computing, mean quantitative figures for the use of computing resources: time, storage space, equipment.

521. **Performance requirements:** By performance requirements we mean *requirements*[605] which express *performance*[520] properties (desiderata).

522. **Petri net:** The Petri net language is a special graphic notation for expressing concurrency of actions, and simultaneity of events, of processes. (See [200].)

523. **Phase:** By a phase we shall here, in the context of software development, understand either the *domain*[239] *development*[228] phase, the *requirements*[605] *development*[228] phase, or the *software design*[688] phase.

524. **Phenomenon:** By a phenomenon we shall mean a physically manifest "thing". (Something that can be sensed by humans (seen, heard, touched, smelled or tasted), or can be measured by physical apparatus: Electricity (voltage, current, etc.), mechanics (length, time and hence velocity, acceleration, etc.), chemistry, etc.)

525. **Phenomenology:** Phenomenology is the study of structures of consciousness as experienced from the first-person point of view [235].

526. **Platform:** By a platform, we shall, in the context of computing, understand a *machine*[436]: Some computer (i.e., hardware) equipment and some software systems. (Typical examples of platforms are: `Microsoft Windows` running on an `IBM ThinkPad Series T` model, or `Trusted Solaris` operating system with an `Oracle Database` 10*g* running on a `Sun Fire E25K Server.`)

527. **Platform requirements:** By platform requirements we mean *requirements*[605] which express *platform*[526] properties (desiderata). (There can be several platform requirements: One set for the platform on which software shall be developed. Another set for the platform(s) on which software shall be utilised. A third set for the platform on which software shall be demonstrated. And a fourth set for the platform on which software shall be maintained. These platforms need not always be the same.)

528. **Pointer:** A pointer is the same as an *address*[22], a *link*[425], or a *reference*[587]: something which refers to, i.e., designates something (typically something else).

529. **Polymorphic:** Polymorphy is a concept associated with functions and the type of the values to which the function applies. If, as for the length of a list function, **len**, that function applies to lists of elements of any type, then we say the **len**gth function is polymorphic. So, in general, the ability to appear in many forms; the quality or state of being able to assume different forms. From Wikipedia, the Free Enclycopedia [228]:

> In computer science, polymorphism is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations. The concept of polymorphism applies to functions as well as types: A function that can evaluate to and be applied to values of different types is known as a polymorphic function. A data type that contains elements of an unspecified type is known as a polymorphic data type. There are two fundamentally different kinds of polymorphism: If the range of actual types that can be used is finite and the combinations must be specified individually prior to use, it is called *ad hoc polymorphism*[23]. If all code is written without mention of any specific type and thus can

be used transparently with any number of new types, it is called *parametric polymorphism*. Programming using the latter kind is called *generic programming*, particularly in the object-oriented community. However, in many statically typed functional programming languages the notion of parametric polymorphism is so deeply ingrained that most *programmers* simply take it for granted.

530. **Portability:** Portability is a concept associated with *software*[685], more specifically with the *program*[545]s (or *data*[193]). Software is (or files, including *database*[195] records, are) said to be portable if it (they), with ease, can be "ported" to, i.e., made to "run" on, a new *platform*[526] and/or compile with a different compiler, respectively different database management system.

531. **Post-condition:** The concept of post-condition is associated with function application. The post-condition of a function f is a predicate p_{o_f} which expresses the relation between argument a and result r values that the function f defines. If a represent argument values, r corresponding result values and f the function, then $f(a) = r$ can be expressed by the post-condition predicate p_{o_f}, namely, for all applicable a and r the predicate p_{o_f} expresses the truth of $p_{o_f}(a, r)$. (See also *pre-condition*[535].)

532. **Postfix:** The concept of postfix is basically a syntactic one, and is associated with operator/operand expressions. It is one about the displayed position of a unary (i.e., a monadic) operator with respect to its operand (expression). An expression is said to be in postfix form if a monadic operator is shown, is displayed, after the expression to which it applies. (Typically the factorial operator, say !, is shown after its operand expression, viz. 7!.)

533. **Post-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First, for each subtree, a subtree post-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is post-order traversed, then just that tree's root is visited (and that tree has then been post-order traversed) and (the leaf) is "marked" as having been post-order visited. After each subtree visit the root of the tree of which the subtree is a subtree is revisited and now it is "marked" as having been visited. (Cf. Fig. 13 on page 233: A left to right post-order traversal of that tree yields the following sequence of "markings": CQXFLUJMZKA; cf. also Fig. 11).

534. **Pragmatics:** Pragmatics is the (i) study and (ii) practice of the factors that govern our choice of language in social interaction and the effects of our choice on others. (We use the term pragmatics in connection with the use of language, as complemented by the *semantics*[655] and *syntax*[733] of language.)

535. **Pre-condition:** The concept of pre-condition is associated with function application where the function being applied is a partial function. That is: for some arguments of its definition set the function yields **chaos**, that is, does not terminate. The

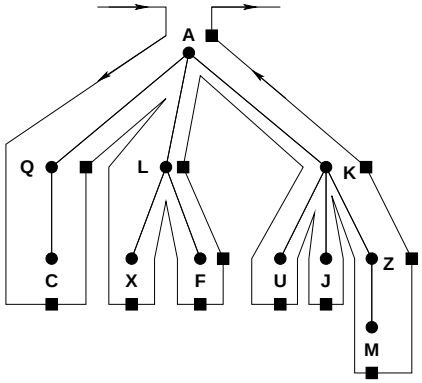

Figure 11: A left to right post-order tree traversal

pre-consition of the function is then a predicate which expresses those values of the arguments for which the function application terminates, that is, yields a result value. (See *weakest pre-condition*[811].)

536. **Predicate:** A predicate is a truth-valued expression involving terms over arbitrary values, well-formed formula relating terms and with *Boolean*[93] *connective*[167]s and *quantifier*[569]s.

537. **Predicate logic:** A predicate logic is a language of *predicate*[536]s (given by some *formal*[296] *syntax*[733]) and a *proof system*[557].

538. **Pre-order:** A special order of *tree traversal*[778] in which visits are made to nodes of trees and subtrees as follows: First to the root of the tree with that root now being "marked" as having been pre-order visited. Then for each subtree a subtree pre-order traversal is made, in the order left to right (or right to left). When a tree, whose number of subtrees is zero, is pre-order traversed, then just that tree's root is visited (and that tree has then been pre-order traversed) and the leaf is then "marked" as having been pre-order visited. (Cf. Fig. 13 on page 233: A right-to-left pre-order traversal of that tree yields the following sequence of "markings": AKZMJULFXQC. Cf. also Fig. 12 on the following page).

539. **Presentation:** By presentation we mean the syntactic *document*[237]ation of the results of some *development*[228].

540. **Prescription:** A prescription is a specification which prescribes something designatable, i.e., which states what shall be achieved. (Usually the term 'prescription' is used only in connection with *requirements*[605] prescriptions.)

541. **Preventive maintenance:** By preventive maintenance — of a *machine*[436] — we mean that a set of special tests are performed on that *machine*[436] in order to ascertain whether the *machine*[436] needs *adaptive maintenance*[21], and/or *corrective main-*

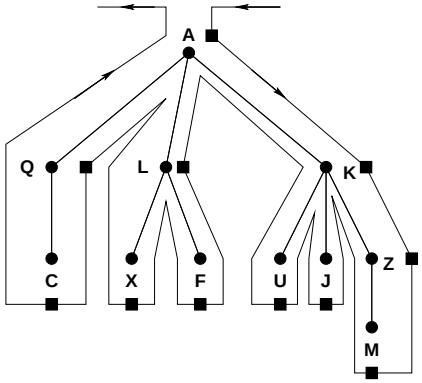

Figure 12: A right-to-left pre-order tree traversal

tenance[187], and/or *perfective maintenance*[519]. (If so, then an update, as here, of software, has to be made in order to achieve suitable *integrity*[388] or *robustness*[631] of the *machine*[436].)

542. **Principle:** An accepted or professed rule of action or conduct, ..., a fundamental doctrine, right rules of conduct, ... [215]. (The concept of principle, as we bring it forth, relates strongly to that of *method*[456]. The concept of principle is "fluid". Usually, by a method, some people understand an orderliness. Our definition puts the orderliness as part of overall principles. Also, one usually expects analysis and construction to be efficient and to result in efficient artifacts. Also this we relegate to be implied by some principles, techniques and tools.)

543. **Procedure:** By a procedure we mean the same as a *function*[310]. (Same as *routine*[636] or *subroutine*[723].)

544. **Process:** By a process we understand a sequence of actions and events. The events designate interaction with some environment of the process.

545. **Program:** A program, in some *programming language*[551], is a formal text which can be subject to *interpretation*[397] by a computer. (Sometimes we use the term *code*[118] instead of program, namely when the program is expressed in the machine language of a computer.)

546. **Programmable:** An *active*[14] *dynamic*[260] *phenomenon*[524] has the programmable (active dynamic) attribute if its *action*[12]s (hence *state*[705] changes) over a future time interval can be accurately prescribed. (Cf. *autonomous*[73] and *biddable*[85].)

547. **Programmer:** A person who does *software design*[688].

548. **Program point:** By a program point we shall here understand any point in a program text (whether of an *applicative programming language*[49] (i.e., *functional pro-*

gramming language[314]), an *imperative programming language*[354], or a *logic programming language*[434]) between any two textually neighbouring *token*[762]s. (The idea of a program point is the following: Assume an *interpreter*[398] of programs of the designated kind. Such an interpreter, at any step of its *interpretation*[397] *process*[544], can be thought of as interpreting a special token, or a sequence of neighbouring tokens, in both cases: "between two program points".)

549. **Program organisation:** By program organisation we loosely mean how a *program*[545] (i.e., its text) is structured into, for example, *module*[464]s (eg., *class*[114]es), *procedure*[543]s, etc.

550. **Programming:** The act of constructing *program*[545]s. From [108]:

> *1: The art of debugging a blank sheet of paper (or, in these days of online editing, the art of debugging an empty file). 2: A pastime similar to banging one's head against a wall, but with fewer opportunities for reward. 3: The most fun you can have with your clothes on (although clothes are not mandatory).*

551. **Programming language:** A language for expressing *program*[545]s, i.e., a language with a precise *syntax*[733], a *semantics*[655] and some textbooks which provides remnants of the *pragmatics*[534] that was originally intended for that programming language. (See next entry: *programming language type*[552].)

552. **Programming language type:** With a *programming language*[551] one can associate a *type*[782]. Typically the name of that type intends to reveal the type of a main paradigm, or a main data type of the language. (Examples are: *functional programming language*[314] (major data type is functions, major operations are definition of functions, application of functions and composition of functions), *logic programming language*[434] (major kinds of expressions are ground terms in a Boolean algebra, propositions and predicates), *imperative programming language*[354] (major kinds of language constructs are declaration of assignable variables, and assignment to variables, and a more or less indispensable kind of data type is references [locations, addresses, pointers]), and *parallel programming language*[507].)

553. **Projection:** By projection we shall here, in a somewhat narrow sense, mean a technique that applies to *domain description*[243]s and yields *requirements prescription*[615]s. Basically projection "reduces" a domain description by "removing" (or, but rarely, *hiding*[337]) *entities*[272], *function*[310]s, *event*[281]s and *behaviour*[79]s from the domain description. (If the domain description is an informal one, say in English, it may have expressed that certain entities, functions, events and behaviours *might* be in (some instantiations of) the domain. If not "projected away" the similar, i.e., informal requirements prescription will express that these entities, functions, events and behaviours *shall* be in the domain and hence *will* be in the environment of the *machine*[436] being requirements prescribed.)

554. **Proof:** A *proof* of a theorem, ϕ, from a set, Γ, of sentences of some *formal*[296] *proposition*[560]al or *predicate*[536] language, $\mathcal{L}$, is a finite sequence of sentences, ϕ_1, ϕ_2, ..., ϕ_n, where $\phi = \phi_1$, where ϕ_n = **true**, and in which each ϕ_i is either an *axiom*[75] of $\mathcal{L}$, or a member of Γ, or follows from earlier ϕ_j's by an *inference rule*[369] of $\mathcal{L}$.

555. **Proof obligation:** A clause of a program may only be (dynamically) well-defined if the values of clause parts lie in certain ranges (viz. no division by zero). We say that such clauses raise proof obligations, i.e., an obligation to prove a property. (Classically it may not be statically (i.e., compile time) checkable that certain expression values lie within certain *subtype*[724]s. Discharging a proof may help ensure such constraints.)

556. **Proof rule:** Same as *inference rule*[369] or *axiom*[75].

557. **Proof system:** A *consistent*[168] and (relative) *complete*[129] set of *proof rule*[556]s.

558. **Property:** A quality belonging and especially peculiar to an individual or thing; an *attribute*[69] common to all members of a class. (Hence: "Not a property owned by someone, but a property possessed by something".)

559. **Property-oriented:** A specification (description, prescription) is said to be property-oriented if the specification (etc.) expresses *attribute*[69]s. (Contrast to *model oriented*[461].)

560. **Proposition:** An expression in language which has a truth value.

561. **Protocol:** A set of formal rules describing how to exchange messages, between a human user and a *machine*[436], or, more classically, across a network. (Low-level protocols define the electrical and physical standards to be observed, bit and byte ordering, and the transmission and error detection and correction of the bit stream. High-level protocols deal with the data formatting, including the syntax of messages, the terminal-to-computer dialogue, character sets, sequencing of messages, etc.)

562. **Pure functional programming language:** A *functional programming language*[314] is said to be pure if none of its constructs designates *side-effects*.

563. **Pushdown stack:** A pushdown stack is a simple *stack*[700]. (Usually a simple stack has just the following operations: *push* an element onto the stack, *pop* the top element from the stack, and observe the *top* element of the stack.)

564. **Pushdown automaton:** A pushdown automaton is an *automaton*[71] with the addition of a *pushdown stack*[563] such that (i) the pushdown automaton *input*[382] is provided both from an environment external to the pushdown automaton and from the *top* of the pushdown stack, (ii) the pushdown automaton *output*[502] is provided to the pushdown stack by being *push*ed onto the top of that stack, and (iii) such

that the pushdown automaton may direct an element to be *pop*ped from the pushdown stack. (The pushdown automaton still has the notion of the final states of the *automaton*[71].)

565. **Pushdown machine:** A pushdown (stack) machine is like a *pushdown automaton*[564] with the addition that now the pushdown machine also provides *output*[502] to the environment of the pushdown machine.

566. **Putative:** Commonly accepted or supposed, that is, assumed to exist or to have existed. (See also *imperative*[352], *indicative*[362] and *optative*[499].)

. . . . . $\mathcal{Q}$

567. **Quality:** Specific and essential character. (Quality is an *attribute*[69], a *property*[558], a characteristic (something has character).)

568. **Quantification:** The operation of quantifying. (See *quantifier*[569]. The x (the y) is quantifying expression $\forall x{:}X{\cdot}P(x)$ (respectively $\exists y{:}Y{\cdot}Q(y)$).)

569. **Quantifier:** A marker that quantifies. It is a prefixed operator that binds the variables in a logical formula by specifying their possible range of *value*[802]s. (Colloquially we speak of the **universal** and the **existential** quantifiers, $\forall$, respectively $\exists$. Typically a quantified expression is then of either of the forms $\forall x{:}X{\cdot}P(x)$ and $\exists y{:}Y{\cdot}Q(y)$. They 'read': For all quantities x of type X it is the case that the predicate $P(x)$ holds; respectively: There exists a quantity y of type Y such that the predicate $Q(y)$ holds.)

570. **Quantity:** An indefinite *value*[802]. (See the *quantifier*[569] entry: The quantities in $P(x)$ (respectively $Q(y)$) are of type X (respectively Y). y is indefinite in that it is one of the quantities of Y, but which one is not said.)

571. **Query:** A request for information, generally as a formal request to a *database*[195].

572. **Query language:** A *formal*[296] *language*[417] for expressing queries (cf. *query*[571]). (The most well-known query language, today, 2005, is `SQL` [77].)

573. **Queue:** A queue is an *abstract data type*[4] with a queue data structure and, typically, the following operations: enqueue (insert into one end of the queue), dequeue (remove from the other end of the queue). Axioms then determine specific queue properties. ()

. . . . . $\mathcal{R}$

574. **Radix:** In a positional representation of numbers, that integer by which the significance of one digit place must be multiplied to give the significance of the next higher digit place. (Conventional decimal numbers are radix ten, binary numbers are radix two.)

575. **RAISE:** RAISE stands for Rigorous Approach to Industrial Software Engineering. (RAISE refers to a method, The RAISE Method [112, 31, 33, 34], a specification language, RSL [110], and "comes" with a set of tools.)

576. **Range:** The concept of range is here used in connection with functions. Same as *range set*[577]. See next entry.

577. **Range set:** Given a *function*[310], its range set is that set of *value*[802]s which is yielded when the function is *applied* to each member of its *definition set*[211].

578. **Reactive:** A *phenomenon*[524] is said to be reactive if the phenomenon performs *action*[12]s in response to external stimuli. Thus three properties must be satisfied for a system to be of reactive dynamic attribute: (i) An interface must be definable in terms of (ii) provision of input stimuli and (iii) observation of (state) reaction. (Contrast to *inert*[367] and *active*[14].)

579. **Reactive system:** A *system*[736] whose main phenomena are chiefly *reactive*[578]. (See the *reactive*[578] entry just above.)

580. **Real time:** We say that a *phenomenon*[524] is real time if its behaviour somehow must guarantee a response to an external event within a given time. (Cf. *hard real time*[330] and *soft real time*[684].)

581. **Reasoning:** Reasoning is the ability to *infer*[368], i.e., to make *deduction*[205]s or *induction*[364]s. (Automated reasoning is concerned with the building and use of computing systems that automate this process. The overall goal is to mechanise different forms of reasoning.)

582. **Recogniser:** A recogniser is an *algorithm*[31] which can decide whether a string can be *generate*[321]d by a given *grammar*[325] of a *language*[417]. (Typically a recogniser can be abstractly formulated as a *finite state automaton*[289] for a *regular language*[594], and as a *pushdown automaton*[564] for a *context-free language*[174].)

583. **Recognition rule:** A recognition rule is a text which describes some *phenomenon*[524], that is, a possibly singleton *class*[114] of such (i.e., their embodied *concept*[152], i.e., *type*[782]), such that it is uniquely decidable, by a human, whether a phenomenon satisfies the rule or not, i.e., is a member of the class, or not. (The recognition rule concept used here is due to Michael A. Jackson [139].)

584. **Recursion:** Recursion is a concept associated both with the *function definition*[316]s and with *data*[193] *type definition*[785]s. A function definition [a data type] is said to possess recursion if it is defined in terms of itself. (Cf. with the slightly different concept of *recursive*[585].)

585. **Recursive:** Recursive is a concept associated with *function*[310]s. A function is said to be recursive if, in the course of the evaluation of an invocation of the function,

that function is repeatedly invoked. (Cf. with the slightly different concept of *recursion*[584].)

586. **Reengineering:** By reengineering we shall, in a narrow sense, only consider the reengineering of business processes. Thus, to us, reengineering is the same as *business process reengineering*[101]. (Reengineering is also used in the wider sense of a major change to some already existing engineering *artefact*[55].)

587. **Reference:** A reference is the same as an *address*[22], a *link*[425], or a *pointer*[528]: something which refers to, i.e., designates something (typically something else).

588. **Referential transparency:** A concept which is associated with certain kinds of *programming*[550] or *specification language*[699] constructs, namely those whose *interpretation*[397] does not entail *side effects*. (A *pure functional programming language*[562] is said to be referentially transparent.)

589. **Refinement:** Refinement is a *relation*[599] between two *specification*[698]s: One specification, D, is said to be a refinement of another specification, S, if all the properties that can be observed of S can be observed in D. Usually this is expressed as $D \sqsubseteq S$. (Set-theoretically it works the other way around: in $D \supseteq S$, D allows behaviours not accounted for in S.)

590. **Refutable assertion:** A refutable assertion is an assertion that might be refuted (i.e., convincingly shown to be false). (Einstein's theory of relativity, in a sense, refuted Newton's laws of mechanics. Both theories amount to assertions.)

591. **Refutation:** A refutation is a statement that (convincingly) refutes an assertion. (Lakatos [147] drew a distinction between refutation (evidence that counts against a theory) and rejection (deciding that the original theory has to be replaced by another theory). We can still use Newton's theory provided we stay within certain boundaries, within which that theory is much easier to handle than Einstein's theory.)

592. **Regular expression:** To introduce the notion of regular expression we assume an *alphabet*[34], A, say finite. Basis clause: For any a in the alphabet, a is a regular expression. Inductive clause: If r and r' are regular expressions, then so are rr', (r), $r \mid r'$, and $r^\star$. (The denotation, $\mathcal{L}(r)$, of a regular expression r is defined as follows: (i) If r is of the form a, for a in the alphabet A, then $\mathcal{L}(a) = \{a\}$; (ii) if r is of the form $r'r''$ then $\mathcal{L}(r'r'') = \{s'^\frown s'' \mid s' \in \mathcal{L}(r'), s'' \in \mathcal{L}(r'')\}$; (iii) or if r is of the form (r') then $\mathcal{L}((r')) = \{s \mid s \in \mathcal{L}(r')\}$; (iv) or if r is of the form $r' \mid r''$ then $\mathcal{L}(r' \mid r'') = \{s \mid s \in \mathcal{L}(r') \vee s \in \mathcal{L}(r'')\}$; (v) or if r is of the form $r'^\star$ then $\mathcal{L}(r'^\star) = \{s \mid s =<> \vee s \in \mathcal{L}(r') \vee s' \in \mathcal{L}(r'r') \vee s' \in \mathcal{L}(r'r'r') \vee \ldots\}$ where $<>$ is the empty string, idempotent under concatenation.)

593. **Regular grammar:** See *regular syntax*.

594. **Regular language:** By a regular language we understand a *language*[417] which is the denotation of a *regular expression*[592]. (Some simple forms of *grammar*[325]s, that is, *regular syntax*[596]es, also generate regular languages.)

595. **Regulation:** A regulation stipulates that an *action*[12] be taken in order to remedy a previous action which "broke" a *rule*[638]. That is, a regulation is some text which designates a possibly composite *action*[12] which, in turn, denotes a state-to-state change which ostensibly results in a result state in which the rule now holds. Usually a domain regulation is paired with domain rule.

596. **Regular syntax:** A regular syntax is a *syntax*[733] which denotes (i.e., which *generate*[321]s) a *regular language*[594].

597. **Reification:** The result of a *reify*[598] action. (See also *data reification*[198], *operation reification*[497] and *refinement*[589].)

598. **Reify:** To regard (something *abstract*[1]) as a material or *concrete*[154] thing. (Our use of the term is more *operational*[494]: To take an *abstract*[1] thing and turn it into a less abstract, more *concrete*[154] thing.)

599. **Relation:** By a relation we usually understand either a mathematical *entity*[272] or an *information structure*[374] consisting of a set of (relation) tuples (like rows in a *table*[739]). The mathematical entity, a relation, can be thought of, also, as a possibly infinite set of n-groupings (i.e., *Cartesian*[107]s of the same *arity*[53]), such that if $(a, b, \cdots, c, d, \cdots, e, f)$ is such an n-tuple, then we may say that $(a, b, \cdots, c)$ (a relation argument) relates to $(d, \cdots, e, f)$ (a relation result). Thus *function*[310]s are special kinds of relations, namely where every argument relates to exactly one result. (Relations, as information structures, are well-known in *relational database*[600]s.)

600. **Relational database:** A *database*[195] whose *data*[193] *types* are (i) *atomic*[63] *values*, (ii) *tuples* of these, and *relations* seen as sets of *tuples*. (The relational database model is due to E.F. Codd [68].)

601. **Reliability:** A system being *reliable* — in the context of a machine being dependable — means some measure of continuous correct service, that is: Measure of time to *failure*[286]. (Cf. *dependability*[217] [being dependable].) (Reliability is a *dependability requirement*[218]. Usually reliability is considered a *machine*[436] property. As such, reliability is (to be) expressed in a *machine requirements*[438] document.)

602. **Renaming:** By renaming we mean *Alpha-renaming*[35]. (Renaming, in this sense, is a concept of the *Lambda-calculus*[412].)

603. **Rendezvous:** Rendezvous is a concept related to parallel processes. It stands for a way of synchronising a number, usually two, of processes. (In CSP the pairing of output (!) / input (?) clauses designating the same channel provides a language construct for rendezvous.)

604. **Representation abstraction:** By *representation abstraction* of [typed] values we mean a specification which does not hint at a particular data (structure) model, that is, which is not implementation biased. (Usually a representation abstraction (of data) is either *property oriented*[559] or is *model oriented*[461]. In the latter case it is then expressed, typically, in terms of mathematical entities such as sets, Cartesians, lists, maps and functions.)

605. **Requirements:** A condition or capability needed by a user to solve a problem or achieve an objective [134].

606. **Requirements acquisition:** The gathering and enunciation of *requirements*[605]. (Requirements acquisition comprises the activities of preparation, requirements *elicitation*[265] (i.e. *requirements capture*[608]) and preliminary requirements evaluation (i.e., requirements vetting).)

607. **Requirements analysis:** By *requirements analysis* we understand a reading of requirements acquisition *rough-sketch*[633] prescription units, (i) with the aim of forming concepts from these requirements prescription units, (ii) as well as with the aim of discovering inconsistencies, conflicts and incompleteness within these requirements prescription units, and (iii) with the aim of evaluating whether a requirements can be objectively shown to hold, and if so what kinds of tests (etc.) ought be devised.

608. **Requirements capture:** By requirements capture we mean the act of eliciting, of obtaining, of extracting, requirements from *stakeholder*[703]s. (For practical purposes requirements capture is synonymous with *requirements elicitation*[611].)

609. **Requirements definition:** Proper *definition*[210]al part of a *requirements prescription*[615].

610. **Requirements development:** By requirements development we shall understand the *development*[228] of a *requirements prescription*[615]. (All aspects are included in development: *requirements acquisition*[606], requirements *analysis*[39], requirements *model*[460]ling, requirements *validation*[800] and requirements *verification*[807].)

611. **Requirements elicitation:** By requirements elicitation we mean the actual extraction of *requirements*[605] from *stakeholder*[703]s.

612. **Requirements engineer:** A requirements engineer is a *software engineer*[692] who performs *requirements engineering*[613]. (Other forms of *software engineer*[692]s are *domain engineer*[247]s and *software design*[688]ers (cum *programmer*[547]).)

613. **Requirements engineering:** The engineering of the development of a *requirements prescription*[615], from identification of *requirements*[605] *stakeholder*[703]s, via *requirements acquisition*[606], *requirements analysis*[607], and *requirements prescription*[615] to requirements *validation*[800] and requirements *verification*[807].

614. **Requirements facet:** A requirements facet is a view of the requirements — "seen from a *domain description*[243]" — such as *domain projection*[255], *domain determination*[245], *domain instantiation*[253], *domain extension*[249], *domain fitting*[251] or *domain initialisation*[252].

615. **Requirements prescription:** By a *requirements*[605] *prescription*[540] we mean just that: the prescription of some requirements. (Sometimes, by requirements prescription, we mean a relatively complete and consistent specification of all requirements, and sometimes just a *requirements prescription unit*[616].)

616. **Requirements prescription unit:** By a *requirements*[605] *prescription*[540] unit we understand a short, "one or two liner", possibly *rough-sketch*[633], *prescription*[540] of some property of a *domain requirements*[258], an *interface requirements*[394], or a *machine requirements*[438]. (Usually requirements prescription units are the smallest textual, sentential fragments elicited from requirements *stakeholder*[703]s.)

617. **Requirements specification:** Same as *requirements prescription*[615] — the preferred term.

618. **Requirements unit:** By a requirements unit we mean a single sentence, i.e., a short expression of a "singular" *requirements*[605]. (A "full" (or complete) *requirements*[605] thus consists of (usually very many) *requirements unit*[618]s.)

619. **Requirements validation:** By requirements validation we rather mean the *validation*[800] of a *requirements prescription*[615].

620. **Resource:** From Old French *ressourse relief, resource, from resourdre to relieve, literally, to rise again, from Latin resurgere . . . an ability to meet and handle a situation* [214] (being resourceful). (In computing we deal with computing resources such as *storage*[715], *time*[761] and further computing equipment. Many computing applications handle enterprise resources such as enterprise staff, production equipment, building or land space, production time, etc. In enterprise domains resources include monies, people, equipment, buildings, time and locations (geographical space).)

621. **Resource allocation:** The *allocation* of *resource*[620]s.

622. **Resource scheduling:** The *scheduling*[646] of *resource*[620]s.

623. **Retrieval:** Used here in two senses: The general (typically *database*[195]-oriented) sense of 'the retrieval [the fetching] of data (of obtaining information) from a repository of such'. And the special sense of 'the retrieval of an abstraction from a concretisation', i.e., abstracting a concept from a phenomenon (or another, more operational concept). (See the next entry for the latter meaning.)

624. **Retrieve function:** By a *retrieve function*[310] we shall understand a function that applies to *values* of some *type*[782], the "more concrete, operational" type, and yields *values* of some *type*[782] claimed to be more *abstract*[1]. (Same as *abstraction function*[5].)

625. **Rewrite:** The replacement of some text or structure by some other text, respectively structure. (See *rewrite rule*[626].)

626. **Rewrite rule:** A rewrite rule is a directed equation: *lhs* = *rhs*. The left- and right-hand sides are *patterns*. If some *text* can be decomposed into three parts, i.e., $text_0 = text_1 \hat{} text_2 \hat{} text_3$, where $text_1$ and/or $text_3$ may be empty texts, and where $text_2$ = *lhs*, then an application of the rewrite rule *lhs* = *rhs* to $text_0$ yields $text_1 \hat{} rhs \hat{} text_3$. (The equation *lhs* = *rhs* is said to be directed in that this rule does not prescribe that a subtext equal to *rhs* is to be rewritten into *lhs*.)

627. **Rewrite system:** Rewrite systems are sets of *rewrite rule*[626]s used to compute, by repeatedly replacing subterms of a given formula with equal terms, until the simplest form possible is obtained [79]. (Rewrite systems form a both theoretically and practically interesting subject. They abound in instrumenting *theorem proving*[758], and the *interpretation*[397] of notably *algebraic semantics*[27] *specification language*[699]s, cf. `CafeOBJ` [81, 80] and `Maude` [67, 172, 59].)

628. **Rigorous:** Favoring rigor, i.e., being precise.

629. **Rigorous development:** Same as the composed meaning of the two terms *rigorous*[628] and *development*[228]. (We usually speak of a spectrum of development modes: *systematic development*[737], rigorous development and *formal development*[298]. Rigorous software development, to us, "falls" somewhere between the two other modes of development: (Always) complete *formal specification*[304]s are constructed, for all (phases and) stages of development; some, but usually not all *proof obligation*[555]s are expressed; and usually only a few are discharged (i.e., proved to hold).)

630. **Risk:** The Concise Oxford Dictionary [160] defines risk (noun) in terms of a hazard, chance, bad consequences, loss, etc., exposure to mischance. Other characterisations of the term risk are: someone or something that creates or suggests a hazard, and possibility of loss or injury.

631. **Robustness:** A *system*[736] is robust — in the context of a *machine*[436] being *dependable* — if it retains all its *dependability*[217] attributes (i.e., properties) after *failure*[286] and after *maintenance*[442]. (Robustness is (thus) a *dependability requirement*[218].)

632. **Root:** A root is a *node*[479] of a *tree*[777] which is not a sub*tree*[777] of a larger, *embedding* (*embedded*[266]) tree.

633. **Rough-sketch:** See next item.

634. **Rough sketch:** By a rough sketch — in the context of *descriptive software development*[690] *documentation* — we shall understand a *document*[237] text which describes something which is not yet consistent and complete, and/or which may still be too concrete, and/or overlapping, and/or repetitive in its descriptions, and/or with which the describer has yet to be fully satisfied.

635. **Route:** Same as *path*[517].

636. **Routine:** Same as *procedure*[543].

637. **RSL:** RSL stands for the RAISE [112] Specification Language [110, 31, 33, 34]. ()

638. **Rule:** A regulating principle. In the *rules and regulations*[640] facet context of modelling domain rules we shall understand a domain rule as some text whose meaning is a *predicate*[536] over a pair of suitably chosen domain *state*[705]s. We may assume that a domain *action*[12] or a domain *event*[281] takes place in the first of these states and results in the second of these states. If the predicate is true then we say that the rule has been obeyed, otherwise that it has not been obeyed. Usually a *regulation*[595] is attached to the rule. (We use the concept of rules in several different contexts: *rewrite rule*[626], *rule of grammar*[639] and *rules and regulations*[640].)

639. **Rule of grammar:** A grammar is made up of one or more rules. A rule has a (left-hand-side) *definiendum*[207] and a (right-hand-side) *definiens*[208]. The definiendum is usually a single *identifier*[351]. The definiens is usually a possibly empty string of *identifier*[351]s. These identifiers are either *terminal*[750]s or *nonterminal*[484]s. A definiendum identifier is a nonterminal. In a grammar all nonterminals have a defining rule. Those identifiers which do not appear as a definiendum of a rule are thence considered terminals.

640. **Rules and regulations:** By rules and regulations we mean guidelines that are intended to be adhered to by the enterprise staff and enterprise customers (i.e., users, clients) in conducting their "business", i.e., their actions within, and with, the enterprise. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *support technology*[725], *management and organisation*[445] and *human behaviour*[345].)

641. **Run time:** The time (or time interval) during which a software *program*[545] is subject to *interpretation*[397] by a computer. (The term run time is usually deployed in order to distinguish between that concept and the concept of *compile time*[127].)

$\mathcal{S}$

642. **Safety:** By safety — in the context of a *machine*[436] being *dependable* — we mean some measure of continuous delivery of service of either correct service, or incorrect service after benign *failure*[286], that is, measure of time to catastrophic failure. (Safety is a *dependability requirement*[218]. Usually safety is considered a *machine*[436] property. As such safety is (to be) expressed in a *machine requirements*[438] *document*[237].)

643. **Safety critical:** A *system*[736] whose *failure*[286] may cause injury or death to human beings, or serious loss of property, or serious disruption of services or production, is said to be safety critical.

644. **Satisfiable:** A *predicate*[536] is said to be *satisfiable* if it is true for at least one *interpretation*[397]. (In this context think of an interpretation as a *binding*[88] of all *free*[305] *variable*[803]s of the predicate expression to *value*[802]s. Cf. *valid*[799].)

645. **Schedule:** A schedule is a *syntactic composite*[133] *concept*[152]. A schedule is a *prescription*[540] for (usually where and) when some *resources* are to be present, i.e., *information*[373] about being spatially and temporally available. (As such a schedule usually also includes some *allocation*[33] *information*[373].)

646. **Scheduling:** The act of providing, of constructing, a *schedule*[645].

647. **Schema:** A structured framework or plan. (We shall also use the term 'schema' in connection with, i.e., as a *rewrite rule*[626] and some *axioms* that apply to, for example, applicative program texts and rewrite into imperative program texts.)

648. **Scheme:** See *schema*[647].

649. **Scope:** We shall use the term scope in two sufficiently different senses: (1) In *programming*[550] the scope of an *identifier*[351] is the region of a *program*[545] text within which it represents a certain thing. This usually extends from the place where it is declared to the end of the smallest enclosing *block*[89] (begin/end or procedure/function body). An inner block may contain a redeclaration of the same identifier, in which case the scope of the outer declaration does not include (is shadowed, occluded, blocked off or obstructed by) the scope of the inner. (2) We also use the term scope in the context of the degree to which a project *scope* and *span* extends: Scope being the "larger, wider" delineation of what a project "is all about", *span*[697] being the "narrower", more precise extent.

650. **Scope check:** Usually a function performed by a *compiler*[125] concerning the definition (declaration) and places of use of identifiers of *program*[545] texts. (Thus the use of *scope*[649] is that of the first (1) sense of item 649.)

651. **Script:** A plan of action. By a domain script we shall, more specifically, understand the structured, almost, if not outright, formally expressed, wording of *rules and regulations*[640] of behaviour. See also *license*[424] and *contract*[181].

652. **Secure:** To properly define the concept of secure, we first assume the concept of an authorised user. Now, a *system*[736] is said to be secure if an un-authorised user, when supposedly making use of that system, (i) is not able to find out what the system does, (ii) is not able to find out how it does 'whatever' it does do, and (iii), after some such "use", does not know whether he/she knows! (The above characterisation represents an unattainable proposition. As a characterisation it is acceptable. But it does not hint at ways and means of implementing secure systems. Once such a system is believed implemented the characterisation can, however be used as a guide in devising tests that may reveal to which extent the system indeed is secure. Secure

systems usually deploy some forms of authorisation and encryption mechanisms in guarding access to system functions.)

653. **Security:** When we say that a *system*[736] exhibits security we mean that it is *secure*[652]. (Security is a *dependability requirement*[218]. Usually security is considered a *machine*[436] property. As such security is (to be) expressed in a *machine requirements*[438] document.)

654. **Selector:** By a selector (a selector function) we understand a function which is applicable to *values* of a certain, defined, composed *type*[782], and which yields a proper component of that value. The function itself is defined by the *type definition*[785].

655. **Semantics:** Semantics is the study and knowledge [incl. specification] of meaning in language [70]. (We make the distinction between the *pragmatics*[534], the semantics and the *syntax*[733] of languages. Leading textbooks on semantics of programming languages are [78, 116, 202, 206, 219, 229].)

656. **Semantic function:** A semantics function is a function which when applied to *syntactic values* yields their *semantic values*.

657. **Semantic type:** By a semantic type we mean a *type*[782] that defines *semantic values*.

658. **Semiotics:** Semiotics, as used by us, is the study and knowledge of *pragmatics*[534], *semantics*[655] and *syntax*[733] of language(s).

659. **Sensor:** A sensor can be thought of as a piece of *technology*[746] (an electronic, a mechanical or an electromechanical device) that senses, i.e., measures, a physical *value*[802]. (A sensor is in contrast to an *actuator*[17].)

660. **Sentence:** (i) A word, clause, or phrase or a group of clauses or phrases forming a syntactic unit which expresses an assertion, a question, a command, a wish, an exclamation, or the performance of an action, that in writing usually begins with a capital letter and concludes with appropriate end punctuation, and that in speaking is distinguished by characteristic patterns of stress, pitch and pauses; (ii) a mathematical or logical statement (as an equation or a proposition) in words or symbols [214].

661. **Sequential:** Arranged in a sequence, following a linear order, one after another.

662. **Sequential process:** A process is sequential if all its observable actions can be, or are, ordered in sequence.

663. **Server:** By a server we mean a *process*[544] or a *behaviour*[79] which *interact*[391]s with another process or behaviour (i.e., a *client*[116]) in order for the server to perform some *action*[12]s on behalf of the client.

664. **Set:** We understand a set as a mathematical entity, something that is not mathematically defined, but is a concept that is taken for granted. (Thus by a set we understand the same as a collection, an aggregation, of distinct entities. Membership (of an entity) of a set is also a mathematical concept which is likewise taken for granted, i.e., undefined.)

665. **Set theoretic:** We say that something is set theoretically understood or explained if its understanding or explanation is based on *sets*.

666. **Shared action:** By a shared action we mean an action that can only be partly computed by the *machine*[436]. That is, the *machine*[436], in order to complete an action, may have to inquire with the *domain*[239] (in order, say, to extract some measurable, time-varying simple entity attribute value) in order to proceed in its computation.

667. **Shared behaviour:** By a shared behaviour we mean a behaviour many of whose actions and events occur both in the *domain*[239] and, in some encoded form, and in the same squence, in the *machine*[436].

668. **Shared concept:** See *shared phenomenon*[676].

669. **Shared data:** See *shared phenomenon*[676].

670. **Shared data initialisation:** By shared data initialisation we understand an *operation*[493] that (initially) creates a *data structure*[199] that reflects, i.e., models, some *shared phenomenon*[676] in the *machine*[436]. (See also *shared data refreshment*[672].)

671. **Shared data initialisation requirements:** *Requirements* for *shared data initialisation*[670]. (See also *computational data+control requirements*[146], *shared data refreshment requirements*[673], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

672. **Shared data refreshment:** By shared data refreshment we understand a *machine*[436] *operation*[493] which, at prescribed intervals, or in response to prescribed events updates an (originally initialised) *shared data*[669] structure. (See also *shared data initialisation*[670].)

673. **Shared data refreshment requirements:** *Requirements* for *shared data refreshment*[672]. (See also *computational data+control requirements*[146], *shared data initialisation requirements*[671], *man-machine dialogue requirements*[447], *man-machine physiological requirements*[448], and *machine-machine dialogue requirements*[437].)

674. **Shared event:** By a shared event we mean an event whose occurrence in the *domain*[239] need be communicated to the *machine*[436] – and, vice-versa, an event whose occurrence in the *machine*[436] need be communicated to the *domain*[239].

675. **Shared information:** See *shared phenomenon*[676].

676. **Shared phenomenon or concept:** A shared phenomenon (or concept) is a phenomenon (respectively a concept) which is present in some *domain*[239] (say in the form of facts, *knowledge*[407] or *information*[373]) and which is also represented in the *machine*[436] (say in the form of some *entity*[272], simple, action, evemt or behaviour). A phenomenon of a domain, when shared, becomes a concept of the machine.

677. **Shared simple entity:** By a shared simple entity we mean a simple entity which both occurs in the *domain*[239] (as a phenomenon or a concept) and in the *machine*[436]. Simple entities that are shared between the domain and the machine must initially be input to the machine. Dynamically arising simple entities must likewise be input and all such machine entities must have their attributes updated, when need arise. Requirements for shared simple entities thus entail requirements for their representation and for their human/machine and/or machine/machine transfer dialogue.

678. **Side effect:** A language construct that designates the modification of the state of a system is said to be a side-effect-producing construct. (Typical side effect constructs are assignment, input and output. A *programming language*[551] "without side effects" is said to be a *pure functional programming language*[562].)

679. **Sign:** Same as *symbol*[728].

680. **Signature:** See *function signature*[318].

681. **Simple entity:** By a simple entity we shall loosely understand an individual, *static*[708] or *inert*[367] *dynamic*[260] (We shall take the narrow view of a simple entity, being in contrast to an *action*[12], an *event*[281] and a *behaviour*[79]; that simple entities "roughly correspond" to what we shall think of as *value*[802]s. We shall further allow simple entities to be either *atomic*[63] or *composite*[133], i.e., in the latter case having decomposable *subentities*[721]. Simple entities have *attribute*[69]s. Composite entities have *attribute*[69]s, *subentities*[721] and a *mereology*[451], the latter explains how the subentities are formed into the simple entity. We consider *simple entities*[681] to be one of the four kinds of *entities*[272] that the `Triptych` "repeatedly" considers. The other three are: *action*[12]s, *event*[281]s and *behaviour*[79]s. Consideration of these are included in the specification of all *domain facet*[250]s and all *requirements facet*[614]s.)

682. **Simplification:** ()

683. **Simulation:** The imitation of the functioning of one system or process by means of the functioning of another. (Attempting to predict aspects of the behaviour of some system by creating an approximate (mathematical) model of it. This can be done by physical modelling, by writing a special-purpose computer program or using a more general simulation package, probably still aimed at a particular kind of simulation [108].)

684. **Soft real time:** By soft real time we mean a *real time*[580] property where the exact, i.e., absolute timing, or time interval, is only of loose, approximate essence. (Cf., *hard real time*[330].)

685. **Software:** By software we understand not only the code that when "submitted" to a computer enables desired computations to take place, but also all the documentation that went into its development (i.e., its *domain description*[243], *requirements specification*[617], its complete *software design*[688] (all stages and steps of *refinement*[589] and *transformation*[771]), the *installation manual*[386], *training manual*[767], and the *user manual*[798]).

686. **Software component:** Same as *component*[131].

687. **Software architecture:** By a software architecture we mean a first kind of specification of software — after requirements — one which indicates **how** the software is to handle the given requirements in terms of *software components* and their interconnection — though without detailing (i.e., designing) these software components.

688. **Software design:** By software design we shall understand the determination of which *components*, which *modules* and which *algorithms* shall implement the *requirements*[605] — together with all the *documents* that usually make up properly documented *software*[685]. (Software design entails *programming*[550], but programming is a "narrower" field of activity than software design in that programming usually excludes many documentation aspects.)

689. **Software design specification:** The *specification*[698] of a *software design*[688].

690. **Software development:** To us, software development includes all three phases of *software*[685] *development*[228]: *domain development*[246], *requirements development*[610] and *software design*[688].

691. **Software development project:** A *software*[685] development project is a planning, research and development project whose aim is to construct *software*[685].

692. **Software engineer:** A software engineer is an *engineer*[269] who performs one or more of the functions of *software engineering*[693]. (These functions include *domain engineering*[248], *requirements engineering*[613] and *software design*[688] (incl. *programming*[550]).)

693. **Software engineering:** The confluence of the science, logic, discipline, craft and art of *domain engineering*, *requirements engineering* and *software design*.

694. **Sort:** A sort is a collection, a structure, of, at present, further unspecified entities. (That is, same as an *algebraic type*. When we say "at present, further unspecified", we mean that the (values of the) sort may be subject to constraining axioms. When we say "a structure", we mean that "this set" is not necessarily a *set*[664] in the simple sense of mathematics, but may be a collection whose members satisfy certain

interrelations, for example, some *partially ordered set*, some *neighbourhood set* or other.)

695. **Sort definition:** The *definition*[210] of a *sort*[694]. (Usually a sort definition consists of the (introduction of) a type name, some (typically *observer function*[490] and *generator function*[323]) *signatures*, and some *axioms* relating sort *values* and *functions*.)

696. **Source program:** By a source program we mean a *program*[545] (text) in some *programming language*[551]. (The term source is used in contrast to target: the result of compiling a source text for some target *machine*[436].)

697. **Span:** Span is here used, in contrast to *scope*[649], more specifically in the context of the degree to which a project *scope* and *span* extend: Scope being the "larger, wider" delineation of what a project "is all about", *span*[697] being the "narrower", more precise extent.

698. **Specification:** We use the term 'specification' to cover the concepts of *domain description*[243]s, *requirements prescription*[615]s and *software design*[688]s. More specifically a specification is a *definition*[210], usually consisting of many definitions.

699. **Specification language:** By a specification language we understand a *formal*[296] *language*[417] capable of expressing *formal*[296] *specifications*. (We refer to such formal specification languages as: `Alloy` [138], `ASM` [199], `Event B` [2, 4, 60], `CafeOBJ` [80, 81], `RSL` [110, 111], `VDM-SL` [52, 107] and `Z` [210, 211, 234, 126].)

700. **Stack:** A stack is an *abstract data type*[4] with a stack data structure and, typically, the following operations: push (onto the top of the stack), pop (remove from the top of the stack). Axioms then determine specific stack properties. ()

701. **Stack activation:** Generally: The topmost element of a stack. Specifically, when a stack is used to record the local states of blocks of a block-structured programming language's blocks or procedure bodies (they are also blocks), then each stack element, i.e., each stack activation, records such a local state and — what is known as static and dynamic — pointers chain such activations together which correspond to the lexicographic scope of the program, respectively the calling invocation of the blocks. (We refer to Vol. 2, Chap. 16, Sect. 16.6.1 for a thorough treatment of stack activations.)

702. **Stage:** (i) By a development stage we shall understand a set of development activities which either starts from nothing and results in a complete phase documentation, or which starts from a complete phase documentation of stage kind, and results in a complete phase documentation of another stage kind. (ii) By a development stage we shall understand a set of development activities such that some (one or more) activities have created new, externally conceivable (i.e., observable) properties of what is being described, whereas some (zero, one or more) other activities have refined

previous properties. (Typical development stages are: *domain*[239] *intrinsics*[399], *domain*[239] *support technologies*, *domain*[239] *management and organisation*[445], *domain*[239] *rules and regulations*[640], etc., and *domain requirements*[258], *interface requirements*[394], and *machine requirements*[438], etc.)

703. **Stakeholder:** By a *domain*[239] (*requirements*[605], *software design*[688])[18] stakeholder we shall understand a person, or a group of persons, "united" somehow in their common interest in, or dependency on the domain (requirements, software design); or an institution, an enterprise, or a group of such, (again) characterised (and, again, loosely) by their common interest in, or dependency on the domain (requirements, software design). (The three stakeholder groups usually overlap.)

704. **Stakeholder perspective:** By a *stakeholder*[703] perspective we shall understand the, or an, understanding of the *universe of discourse*[793] shared by the specifically identified stakeholder group — a view that may differ from one stakeholder group to another stakeholder group of the same universe of discourse.

705. **State:** By a state we shall, in the context of computer *programs*, understand a summary of past *computations*, and, in the context of *domains*, a suitably selected set of *dynamic*[260] *entities*.

706. **Statechart:** The Statechart language is a special graphic notation for expressing communication between and coordination and timing of processes. (See [120].)

707. **Statement:** We shall take the rather narrow view that a statement is a *programming language*[551] construct which *denotes* a *state*[705]-to-state function. (Pure expressions are then programming language constructs which denote state-to-value functions (i.e., with no *side effect*[678]), whereas "impure" expressions, also called clauses, denote state-to-state-and-value functions.)

708. **Static:** An *entity*[272] is static if it is not subject to *actions* that change its *value*[802]. (In contrast to *dynamic*[260].)

709. **Static semantics:** The concept of static semantics is one that applies to *syntactic entities*, typically *programs* or *specifications* of *programming language*[551]s, respectively *specification language*[699]s. The static semantics of such a language is now a *predicate*[536] that applies to *programs* (respectively *specifications*) and yields true if the *program*[545] (*specification*[698]) is syntactically well formed according to the static semantics criteria, typically that certain relations are satisfied between dispersed parts of the *program*[545] (*specification*[698]) texts.

710. **Static typing:** Enforcement of *type checking* at *compile time*[127]. (A *programming language*[551] (or a *specification language*[699]) is said to be statically typed if its *programs* (resp. *specifications*) can be statically *type checked*.)

[18] These three areas of concern form three *universes of discourse*.

711. **Step:** By a development step we shall understand a refinement of a domain description (or a requirements prescription, or a software design specification) module, from a more abstract to a more concrete description (or a more concrete requirements prescription, or a more concrete software design specification).

712. **Stepwise development:** By a stepwise development we shall understand a *development*[228] that undergoes *phases*, *stages* or *steps* of development, i.e., can be characterised by pairs of two adjoining *phase*[523] *steps*, a last *phase*[523] *step*[711] and a (first) next *phase*[523] *step*[711], or two adjoining *stage*[702] *steps*.

713. **Stepwise refinement:** By a stepwise refinement we understand a pair of adjoining *development*[228] *steps* where the transition from one *step*[711] to the next *step*[711] is characterised by a *refinement*[589]. (Refinement is thus always stepwise refinement.)

714. **Store:** Same as *storage*[715]; see next.

715. **Storage:** By storage we shall understand a *function*[310] from *locations* to *values*. (Thus we emphasise the mathematical character of storage rather than any technological character (such as disk storage, etc.).)

716. **Strategy:** [214]: (1) The science and art of employing the political, economic, psychological, and military forces of a nation or group of nations to afford the maximum support to adopted policies in peace or war; (2) an adaptation or complex of adaptations (as of behaviour or structure) that serves or appears to serve an important function in achieving evolutionary success. (Applied to business enterprises the above "translates" into: the science and art of employing the economic and other resources of an enterprise to achieve maximum support for adopted enterprise policies: enterprise products & service profile, market share, growth, profitability, etc.)

717. **Strict function:** A strict function is a function which yields **chaos** (i.e., is undefined) if any of the function arguments are undefined (i.e., **chaos**). (In RSL the logical connectives are not strict. All other functions, built-in or defined, are strict.)

718. **Strongest post-condition:** See *weakest pre-condition*[811].

719. **Structure:** The term 'structure' is understood rather loosely. Normally we shall understand a structure as a mathematical structure, such as an *algebra*[26], or a *predicate logic*[537], or a *Lambda-calculus*[412], or some defined abstraction (a *scheme*[648] or a *class*[114]). (Set theory is a (mathematical) structure. So are RSL's Cartesian, list and map data types.)

720. **Structural operational semantics:** By a structural operational semantics we understand an *operational semantics*[496] which is expressed in terms of a number of *transition rule*[773]s. (See [196].)

721. **Subentity:** A subentity is a proper part of a (thus) non-*atomic*[63] *entity*[272]. (Do not confuse a subentity of an entity with an *attribute*[69] of that entity (or of that subentity).)

722. **Substitution:** By substitution we mean the replacement of a token (viz.: an identifier) by a structure, usually a text. (The most common form of substitution is that of *Beta-reduction*[84] (in the *Lambda-calculus*[412]). Substitution is a "simpler" form of *rewriting*.)

723. **Subroutine:** Same as *routine*[636].

724. **Subtype:** To speak of a subtype we must first be able to speak of a *type*[782], i.e., colloquially, a (suitably structured) set of *value*[802]s. A subtype of a type is then a (suitably structured) and proper subset of the values of the type. (Usually we shall, in RSL, think of a predicate, p, that applies to all members of the type, T, and singles out a proper subset whose elements satisfy the predicate: $\{a \mid a : T \cdot p(a)\}$.)

725. **Support technology:** By a support technology we understand a *facet*[285] of a *domain*[239], one which reflects its (current) dependency on mechanical, electro-mechanical, electronic and other technologies (i.e., tools) in order to carry out its *business process*[99]es. (Other facets of an enterprise are those of its *intrinsics*[399], *business process*[99]es, *management and organisation*[445], *rules and regulations*[640] and *human behaviour*[345].)

726. **Surjection:** A *surjective function*[727] represents surjection. (See also *bijection*[86] and *injection*[379].)

727. **Surjective function:** A *function*[310] which maps *value*[802]s of its postulated *definition set*[211] into all of its postulated *range set*[577] is called surjective. (See also *bijective function*[87] and *injective function*[380].)

728. **Symbol:** Something that stands for or suggests something else, that is, an arbitrary or conventional sign used in writing.

729. **Synchronisation:** By synchronisation we understand the act of ensuring *synchronism*[730] between occurrence of designated *events* in two or more *processes*. (Usually synchronisation between occurrence of designated events in two or more processes entails the exchange of *information*[373], i.e., *data*[193], between these processes, i.e., *communication*[122].)

730. **Synchronism:** A chronological arrangement of *event*[281]s.

731. **Synchronous:** Happening, existing, or arising at precisely the same *time*[761] indicating *synchronism*[730].

732. **Synopsis:** By a synopsis we shall understand a composition of *informative documentation*[375] and *rough-sketch*[633] *description*[220] of some project.

733. **Syntax:** By syntax we mean (i) the ways in which words are arranged to show meaning (cf. *semantics*) within and between sentences, and (ii) rules for forming *syntactically correct* sentences. (See also *regular syntax*, *context-free syntax*, *context-sensitive syntax* and *BNF* for specifics.)

734. **Synthesis:** The construction of an *artefact*[55].

735. **Synthetic:** Result of *synthesis*[734]: not *analytic*[40].

736. **System:** A regularly interacting or interdependent group of phenomena or concepts forming a whole, that is, a group of devices or artificial objects or an organization forming a network especially for producing something or serving a common purpose. (This book will have its own characterisation of the concept of a system (commensurate, however, with the above encircling characterisation); cf. Vol. 2, Sect. 9.5's treatment of system.)

737. **Systematic development:** Systematic development of software is *formal development "lite"!* (We usually speak of a spectrum of development modes: systematic development, *rigorous development*[629], and *formal development*[298]. Systems software development, to us, is at the "informal" extreme of the three modes of development: *formal specification*[304]s are constructed, but maybe not for all stages of development; and usually no proof obligations are expressed, let alone proved. The three volumes of this series of textbooks in software engineering can thus be said to expound primarily the systematic approach.)

738. **Systems engineering:** By systems engineering we shall here understand computing systems engineering: The confluence of developing *hardware*[331] and *software*[685] solutions to *requirements*[605].

. . . . . $\mathcal{T}$

739. **Table:** By a table we understand an *information structure*[374] which can be thought of as an ordered *list*[428] of rows, each row consisting of an ordered *list*[428] of entries, each consisting of some *information*[373]. (When thought of as a *data structure*[199], a table is normally thought of as either a matrix or a *relation*[599].)

740. **Tangibility:** Noun of *tangible*[742].

741. **Tactic:** [214]: (1) a device for accomplishing an end (2) a method of employing forces in combat.Applied to business enterprises the above "translates" into: a set of resource-dependent actions thought to accomplish a strategy.

742. **Tangible:** Physically manifest. That is, can be humanly sensed: heard, seen, smelled, tasted, or touched, or physically measured by a physical apparatus: length (meter, m), mass (kilogram, kg), time (second, s), electric current (Ampere, A), thermodynamic temperature (Kelvin, K), amount of substance (mole, mol), luminous intensity (candela, cd).

743. **Target program:** The concept of target program stems from the fact that *programs* of ordinary *programming languages* need to be translated into some intermediary language or final machine, i.e., computer hardware, language, before their designated computations (i.e., interpretations) can take place. By a target program we understand such an intermediary or final program. (Besides the final target languages made up from the repertoire of computer hardware instructions and computer (bit, byte, half-word, word, double-word and variable field) data formats, special intermediary languages have been devised: `P-code` [94] (into which `Pascal` programs can be translated) [230, 129, 231, 141, 232, 136, 7], `A-code` [93] (into which `Ada` programs can be translated) [56, 226], etc.)

744. **Taxonomy:** By *taxonomy* is meant [160]: "classification, especially in relation to its general laws or principles; that department of science, or of a particular science or subject, which consists in or relates to classification.".

745. **Technique:** A procedure, an approach, to accomplish something.

746. **Technology:** We shall in these volumes be using the term technology to stand for the results of applying scientific and engineering insight. This, we think, is more in line with current usage of the term IT, information technology.

747. **Temporal:** Of or relating to time, including sequence of time, or to time intervals (i.e., durations).

748. **Temporal logic:** A(ny) *logic*[432] over *temporal*[747] *phenomena*. (We refer to Vol. 2, Chap. 15 for our survey treatment of some temporal logics.)

749. **Term:** From [160]: A word or phrase used in a definite or precise sense in some particular subject, as a science or art; a technical expression. More widely: any word or group of words expressing a notion or conception, or denoting an object of thought. (Thus, in RSL, a term is a *clause*[115], an *expression*[282], a *statement*[707], which has a *value*[802] (statements have the **Unit** value).)

750. **Terminal:** By a terminal we shall mean a terminal *symbol*[728] which (in contrast to a *nonterminal*[484] symbol) designates something specific.

751. **Termination:** The concept of termination is associated with that of an *algorithm*[31]. We say that an algorithm, when subject to *interpretation*[397] (colloquially: 'execution'), may, or may not terminate. That is, may halt, or may "go on forever, forever looping". (Whether an algorithm terminates is *undecidable*[792].)

752. **Terminology:** By terminology is meant ([160]): The doctrine or scientific study of terms; the system of terms belonging to a science or subject; technical terms collectively; nomenclature.

753. **Term rewriting:** Same as *rewriting*.

754. **Test:** A test is a means to conduct *testing*[755]. (Typically such a test is a set of data values provided to a program (or a specification) as values for its *free*[305] *variables*. *Testing* then evaluates the program (resp., interprets (symbolically) the specification) to obtain a result (value) which is then compared with what is (believed to be) the, or a, correct result. See Vol. 3, Sects. 14.3.2, 22.3.2 and 29.5.3 for treatments of the concept of test.)

755. **Testing:** Testing is a systematic effort to refute a claim of correctness of one (e.g., a concrete) specification (for example a program) with respect to another (the abstract) specification. (See Vol. 3, Sects. 14.3.2, 22.3.2, and 29.5.3 for treatments of the concept of testing.)

756. **Theorem:** A theorem is a *sentence*[660] that is *provable* without assumptions, that is "purely" from *axioms* and *inference rules*.

757. **Theorem prover:** A mechanical, i.e., a computerised means for *theorem proving*[758]. (Well-known theorem provers are: `PVS` [184, 185] and `HOL/Isabelle` [180].)

758. **Theorem proving:** The act of *proving theorems*.

759. **Theory:** A formal theory is a *formal*[296] *language*[417], a set of *axioms* and *inference rules* for *sentences* in this language, and is a set of *theorems* proved about sentences of this language using the axioms and inference rules. A mathematical theory leaves out the strict formality (i.e., the *proof*[554]system) requirements and relies on mathematical proofs that have stood the social test of having been scrutinised by mathematicians.

760. **Three-valued logic:** Standard logics are two value: **true** and **false**. A three-valued logic is a logic for which the Boolean connectives accept a third value, usually referred to as the *undefined*, or *chaotic* (non-*termination*[751] of operand *expression*[282] *evaluation*[280]). (There can be, and are, many three-valued logics. `RSL` has one set of definitions of the outcome of Boolean ground term evaluation with **chaos** operands. `LPF` is a logic for partial functions sugggested as a logic for `VDM` [16, 65]. John McCarthy [168] first broached the topic of three-valued logics in computing.)

761. **Time:** Time is often a notion that is taken for granted. But one may do well, or better, in trying to understand time as some point set that satisfies certain axioms. Time and space are also often related (via [other] physically manifest "things"). Again their interrelationship needs to be made precise. (In comparative concurrency semantics one usually distinguishes between linear time and branching time semantic equivalences [223]. We refer to our treatment of time and space in Vol. 2 Chap. 5, to Johan van Benthem's book *The Logic of Time* [222], and to Wayne D. Blizard's paper *A Formal Theory of Objects, Space and Time* [57].)

762. **Token:** Something given or shown as an identity. (When, in `RSL`, we define a *sort*[694] with no "constraining" axioms, we basically mean to define a set of tokens.)

763. **Tool:** An instrument or apparatus used in performing an operation. (The tools most relevant to us, in software engineering, are the *specification*[698] and *programming language*[551]s as well as the *software*[685] packages that aid us in the development of (other) software.)

764. **Topology:** (i) A branch of mathematics concerned with those properties of geometric configurations (as point sets) which are unaltered by elastic deformations (as a stretching or a twisting) that are *homeomorphisms*; (ii) the set of all open subsets of a *topological space* (i.e., being or involving properties unaltered under a homeomorphism [continuity and connectedness are topological properties]) [214].

765. **Total algebra:** A total *algebra* is an algebra all of whose functions are total over the carrier.

766. **Trace:** The concept of trace is linked to the concept of a *behaviour*[79]. Trace is then defined as a sequence of *actions* and *events*. ()

767. **Training manual:** A *document*[237] which can serve as a basis for a (possibly self-study) course in how to use a *computing system*[151]. (See also *installation manual*[386] and *user manual*[798].)

768. **Transaction:** General: A communicative action or activity involving two *agent*[24]s that reciprocally influence each other. (Special: The term transaction has come to be used, in computing, notably in connection with the use of database management systems (DBMS, or similar multiuser systems): A transaction is then a unit of interaction with a DBMS (etc.). To further qualify as being a transaction, it must be handled, by the DBMS (etc.), in a coherent and reliable way independent of other transactions.)

769. **Transduce:** To convert (a physical signal, or a message) into another form.

770. **Transducer:** A device that is actuated by power from one system and supplies power usually in another form to a second system. (*Finite state machines* and *pushdown stack machines* are considered transducers.)

771. **Transformation:** The operation of changing one configuration or expression into another in accordance with a precise rule. (We consider the results of *substitution*[722], of *translation* and of *rewriting* to be transformations of what the *substitution*[722], the *translation* and the *rewriting* was applied to.)

772. **Transition:** Passage from one state, stage, subject or place to another; a movement, development, or evolution from one form, stage or style to another [214].

773. **Transition rule:** A *rule*[638], of such a form that it can specify how any of a well-defined class of *states* of a *machine*[436] may make *transitions* to another state, possibly *nondeterministically* to any one of a well-defined number of other states. (The seminal

1981 report *A Structural Approach to Operational Semantics*, by Gordon D. Plotkin [193], set a de facto standard for formulating transition rules (exploring their theoretical properties and uses).)

774. **Translate:** See *translation*[775].

775. **Translation:** An act, process or instance of translating, i.e., of rendering from one language into another.

776. **Translator:** Same as a *compiler*[125].

777. **Tree:** An *acyclic*[18] un-*directed graph*[232]. Thus a tree (i) has a *root*[632], which is a *node*[479], and (ii) zero, one or more, possibly (*branch*[97] or *edge*[262]) *label*[410]led subtrees. Trees or subtrees with no further subtrees have their roots being equated with leaves. Nodes may be labelled. (This characterisation allows for trees with no labels, with only labelled nodes, with only labelled branches, with labelled nodes and branches, or with only some nodes and some branches being labelled. The characterisation usually is interpreted as only allowing finite trees, but one could dispense of the "finite applicability" of the above (i–ii) clauses, to allow infinite trees. The branch concept, akin to the *edge*[262] concept, amounts, however, to a directed edge, i.e., an *arrow*[54]. We refer specifically to *parse tree*[512]s. See also a "redefinition" of trees as found just below, under *tree traversal*[778], including Fig. 13.)

778. **Tree traversal:** A way of visiting (all) the *node*[479]s of a *tree*[777]. Redefine the notion of a *tree*[777] as just given above: Now a tree is a root node and an ordered set (i.e., like a list) of zero, one or more subtrees; each subtree is a tree. Roots are labelled. Hence subtrees are labelled. A tree with an empty set of subtrees is called a leaf. Their roots are the leaves. A tree traversal is now a way of visiting, in some order, as indicated by the order of subtrees, (all) the nodes: the root, the branch nodes and leaves, of a tree. (See the tree of Fig. 13 on the next page. It will be referred to in entries *in-order*[381], *post-order*[533] and *pre-order*[538].)

779. **Triptych:** An ancient Roman writing tablet with three waxed leaves hinged together; a picture (as an altarpiece) or carving in three panels side by side [214]. (The trilogy of the *phases* of *software development*[690], *domain engineering*[248], *requirements engineering*[613] and *software design*[688] as promulgated by this trilogy of volumes!)

780. **Tuple:** A grouping of values. (Like 2-tuplets, quintuplets, etc. Used extensively, at least in the early days, in the field of relational databases — where a tuple was like a row in a relation (i.e., table).)

781. **Turing machine:** A hypothetical machine defined in 1935–1936 by Alan Turing and used for computability theory proofs. It can be understood as consisting of a *finite state machine*[290] and an infinitely long "tape" with symbols (chosen from some finite set) written at regular intervals. A pointer marks the current position and the

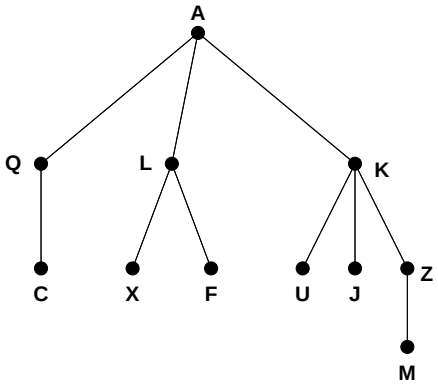

Figure 13: A labelled, ordered tree

machine is in one of states. At each step the machine reads the symbol at the current position on the tape. For each combination of current state and symbol read, the finite state machine specifies the new state and either a symbol to write to the tape or a direction to move the pointer (left or right) or to halt [108]. (Turing machines are equivalent, in computational power, to the *Lambda-calculus*[412].)

782. **Type:** Generally a certain kind of set of *value*s. (See *algebraic type*, *model-oriented type*, *programming language type* and *sort*.)

783. **Type check:** The concept of type check arises from the concepts of *function signatures* and function *arguments*. If arguments are not of the appropriate type then a type check yields an *error*[278] result. (By appropriate *static*[708] *typing*[788] of *declarations* of *variables* of a *programming language*[551] or a *specification language*[699] one can perform static type checking (i.e., at *compile time*[127]).)

784. **Type constructor:** A type constructor is an operation that applies to *types* and yields a *type*[782]. (The type constructors of RSL include the power set constructors: **-set** and **-infset**, the Cartesian constructor: $\times$, the list constructors: * and $^\omega$, the map constructor: $\xrightarrow[m]{}$, the total and partial function space constructors: $\rightarrow$ and $\xrightarrow{\sim}$, the union type constructor: $|$, and others.)

785. **Type definition:** A type definition semantically associates a *type name*[787] with a *type*[782]. Syntactically, as, for example, in RSL, a type definition is either a *sort*[694] definition or is a *definition*[210] whose right-hand side is a *type expression*[786].

786. **Type expression:** A type expression semantically denotes a *type*[782]. Syntactically, as, for example, in RSL, a type expression is an expression involving *type names* and *type constructors*, and, rarely, *terminals*.

787. **Type name:** A type name is usually just a simple *identifier*[351].

788. **Typing:** By typing we mean the association of *types* with *variables*. (Usually such an association is afforded by pairing a *variable*[803] *identifier*[351] with a *type name*[787] in the variable *declaration*[201]. See also *dynamic typing*[261] and *static typing*[710].)

.....$\mathcal{U}$

789. **UML:** Universal Modelling Language. A hodgepodge of notations for expressing requirements and designs of computing systems. (Vol. 2, Chaps. 10, and 12–14 outlines our attempt to "UML"-ize formal techniques.)

790. **Universal algebra:** A universal *algebra*[26] is an *abstract algebra*[2] where we leave the postulates (axioms, laws) unspecified. (The universal level of abstract, the viewpoint of universal algebras, represents for us [159], the high water mark of abstraction in the treatment of *algebraic systems*[28].)

791. **Underspecify:** By an underspecified expression, typically an identifier, we mean one which for repeated occurrences in a specification text always yields the same value, but what the specific value is, is not knowable. (Cf. *nondeterministic*[481] or *loose specification*[435].)

792. **Undecidable:** A formal logic system is undecidable if there is no *algorithm*[31] which prescribes *computation*[144]s that can determine whether any given sentence in the system is a theorem.

793. **Universe of discourse:** That which is being talked about; that which is being discussed; that which is the subject of our concern. (The four most prevalent universes of discourse of this book, this series of volumes on software engineering, are: *software development*[690] *methodology*[457], *domains*, *requirements*[605] and *software design*[688].)

794. **Update:** By an update we shall understand a change of value of a variable, including also the parts, or all, of a *database*[195].

795. **Update problem:** By the update problem we shall understand that data stored in a *database*[195] usually reflect some state of a domain, but that changes in the external state of that domain are not always properly, including timely, reflected in the database.

796. **User:** By a user we shall understand a person who uses a *computing system*[151], or a *machine*[436] (i.e., another computing system) which *interfaces* with the former. (Not to be confused with *client*[116] or *stakeholder*[703].)

797. **User-friendly:** A "lofty" term that is often used in the following context: *"A computing system, a machine, a software package, is required to be user-friendly"* — without the requestor further prescribing the meaning of that term. Our definition of the term user-friendly is as follows: A *machine*[436] (software + hardware) is said to be user-friendly (i) if the *shared phenomena* of the application *domain*[239] (and

machine[436]) are each implemented in a transparent, one-to-one manner, and such that no IT jargon, but common application *domain*[239] *terminology*[752] is used in their (i.1) accessing, (i.2) *invocation*[402] (by a human *user*[796]), and (i.3) display (by the machine); i.e., (ii) if the *interface requirements*[394] have all been carefully expressed (commensurate, in further detailed ways: ..., with the user psyche) and correctly implemented; and (iii) if the machine otherwise satisfies a number of *performance* and *dependability requirements*[605] that are commensurate, in further detailed ways: ..., with the user psyche.

798. **User manual:** A *document*[237] which a regular user of a *computing system*[151] refers to when in doubt concerning the use of some features of that system. (See also *installation manual*[386] and *training manual*[767].)

. . . . . $\mathcal{V}$

799. **Valid:** A *predicate*[536] is said to be *valid* if it is true for all *interpretation*[397]s. (In this context think of an interpretation as a *binding*[88] of all *free*[305] *variable*[803]s of the predicate expression to *value*[802]s; cf. *satisfiable*[644].)

800. **Validation:** (Let, in the following *universe of discourse*[793] stand consistently for either *domain*[239], *requirements*[605] or *software design*[688].) By universe of discourse validation we understand the assurance, with universe of discourse *stakeholders*, that the specifications produced as a result of universe of discourse acquisition, universe of discourse analysis and *concept formation*[153], and universe of discourse domain *modelling* are commensurate with how the stakeholder views the universe of discourse. (*Domain* and *requirements validation*[619] is treated in Vol. 3, Chaps. 14 and 22.)

801. **Valuation:** Same as *evaluation*[280].

802. **Value:** From (assumed) Vulgar Latin *valuta,* from feminine of *valutus,* past participle of Latin *valere* to be of worth, be strong [214]. (Commensurate with that definition, value, to us, in the context of programming (i.e., of software engineering), is whatever mathematically founded *abstraction*[3] can be captured by our *type*[782] and *axiom*[75] *systems*. (Hence numbers, truth values, *tokens*, sets, Cartesians, lists, maps, functions, etc., of, or over, these.))

803. **Variable:** (i) From Latin *variabilis,* from *variare* to vary; (ii) able or apt to vary; (iii) subject to variation or changes [214]. (Commensurate with that definition, a variable, to us, in the context of programming (i.e., of software engineering), is a *placeholder*, for example, a *storage*[715] *location*[431] whose *contents* may change. A variable, further, to us, has a name, the variable's identifier, by which it can be referred.)

804. **VDM:** `VDM` stands for the `Vienna Development Method` [52, 53]. (`VDM-SL` (SL for Specification Language) was the first formal specification language to have an international standard: `VDM-SL, ISO/IEC 13817-1: 1996.` The author of this book

coined the name VDM in 1974 while working with Hans Bekič, Cliff B. Jones, Wolfgang Henhapl and Peter Lucas, on what became the VDM description of PL/I. The IBM Vienna Laboratory, in Austria, had, in the 1960s, researched and developed semantics descriptions [17, 18, 19, 163] of PL/I, a programming language of that time. "JAN" (John A.N.) Lee [154] is believed to have coined the name VDL [155, 162] for the notation (the Vienna Definition Language) used in those semantics definitions. So the letter M follows, lexicographically, the letter L, hence VDM.)

805. **VDM–SL:** VDM-SL stands for the VDM Specification Language. (See entry VDM above. Between 1974 and the late 1980s VDM-SL was referred to by the acronym Meta-IV: the fourth metalanguage (for language definition) conceived at the IBM Vienna Laboratory during the 1960s and 1970s.)

806. **Verb:** A *word*[814] that characteristically is the grammatical centre of a sentence and expresses an act, occurrence or mode of being that in various languages is inflected for agreement with the subject, for tense, for voice, for mood, or for aspect, and that typically has rather full descriptive meaning and characterizing quality but is sometimes nearly devoid of these especially when used as an auxiliary or linking verb [214]. (We shall often find, in modelling, that we model verbs as *functions* (incl. *predicates*).)

807. **Verification:** By verification we mean the process of determining whether or not a specification (a description, a prescription) fulfills a stated property. (That stated property could (i) either be a property of the specification itself, or (ii) that the specification relates, somehow, i.e., is correct with respect to some other specification.)

808. **Verify:** Same, for all practical purposes, as *verification*[807].

809. **Vertex:** Same as an *node*[479].

. . . . . $\mathcal{W}$

810. **Waterfall diagram:** By a waterfall diagram is understood a two-dimensional diagram with a number of boxes placed, say, on a diagonal, from a top left corner of the diagram to a lower right corner, such that the individual boxes are sufficiently spaced apart, i.e., do not overlap, and such that arrows (i.e., "the water") infix adjacent boxes along a perceived diagonal line. (The idea is then that a preceding box, from which an arrow emanates, designates a software development activity that must, somehow, be concluded before activity can start on the software development activity designated by the box upon which the infix arrow is incident.)

811. **Weakest pre-condition:** The condition that characterizes the set of all initial states, such that activation will certainly result in a properly terminating happening leaving the system in a final state satisfying a given post-condition, is called "the weakest pre-condition corresponding to that post-condition". (We call it "weakest",

because the weaker a condition, the more states satisfy it and we aim here at characterising all possible starting states that are certain to lead to a desired final state.)

812. **Well-formedness:** By well-formedness we mean a concept related to the way in which *information*[373] or *data structure*[199] definitions may be given. Usually these are given in terms of *type definition*[785]s. And sometimes it is not possible, due to the *context-free*[173] nature of type definitions. (Well-formedness is here seen separate from the *invariant*[400] over an *information*[373] or a *data structure*[199]. We refer to the explication of *invariant*[400]!)

813. **Wildcard:** A special symbol that stands for one or more characters. (Many operating systems and applications support wildcards for identifying files and directories. This enables you to select multiple files with a single specification. Typical wildcard designators are * (asterisk) and _ (underscore).)

814. **Word:** A speech sound or series of speech sounds or a character or series of juxtaposed characters that symbolizes and communicates a meaning without being divisible into smaller units capable of independent use [214].