Formal Software Specification Using RAISE Exercises

3rd Edition

Anne Elisabeth Haxthausen

January 2002

Contents

1 Introduction

This document, which is an update of [Hax94], contains exercises in formal software specification using RAISE.

The exercises are based on a textbook, [RAI92], on the RAISE specification language, RSL, and extracts of a handbook on RAISE justifications, [GP92]. Hence, the exercises refer to these books.

The exercises in sections 2-17 cover the topics of the RSL textbook (including features such as model-oriented specification, algebraic specification, applicative specification, imperative specification, concurrency, modularity), and section 16 and 17 covers refinement and verification. Section 18 is devoted to exercises in a special application area: language based systems. Finally, in section 19, there is a collection of sample exam questions.

Sources of inspiration for the exercises have been [HP92], [Bjø] and [Jon86].

2 Logic

1. Below are some equivalences which always hold (are true) in classical logic. Which of them hold in RAISE's conditional logic?

   (a) $\sim(\sim a) \equiv a$
   (b) $\mathbf{true} \vee a \equiv \mathbf{true}$
   (c) $a \vee \mathbf{true} \equiv \mathbf{true}$
   (d) $a \Rightarrow b \equiv \sim a \vee b$
   (e) $a \vee \sim a \equiv \mathbf{true}$
   (f) $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$
   (g) $(a \vee b) \vee c \equiv a \vee (b \vee c)$
   (h) $(a = a) \equiv \mathbf{true}$
   (i) $(a \equiv a) \equiv \mathbf{true}$

2. Reduce the following expressions to simpler expressions:

   (a) **if true then false else chaos end** $\equiv$?
   (b) **if** a **then** $\sim$(a $\equiv$ **chaos**) **else false end** $\equiv$?

   Hint:
   Use the following equivalences:

   if true then a **else** b **end** $\equiv$ a (1)

   if false then a **else** b **end** $\equiv$ b (2)

   if a **then** e1 **else** e2 **end** $\equiv$
   if a **then** e1[**true**/a] **else** e2[**false**/a] **end** (3)

3. Which of the following expressions are true:

   (a) $\forall$ i : **Int** • $\exists$ j : **Int** • i + j = 0
   (b) $\forall$ i : **Int** • $\exists$ j : **Nat** • i + j = 0
   (c) $\exists$ i : **Int** • $\forall$ j : **Int** • i + j = 0

4. Write an RSL value expression which expresses the fact that there is not a largest integer.

5. Complete the following definition of a function which tests whether a natural number is even.

   is_even : **Nat** $\rightarrow$ **Bool**
   is_even(n) $\equiv$ ...

3 Products

1. Consider example 5.3 in [RAI92].
   (a) Define a type, 'Circle', with an appropriate representation for circles.
   Hint: You need the following type: $\{|\ r : \mathbf{Real} \bullet r \geq 0.0\ |\}$
   (b) Define a function, 'on_circle', which takes a circle and a position as arguments and checks whether the position is on the arc of the circle.
   (c) Define a value, 'circle', which represents a circle with centre in the origin of the system of coordinates and a radius of length 3.0.
   (d) Define a value, 'pos', representing some (not yet known) position on the arc of the circle ('circle') defined in (c).
2. (a) Define a type, 'Complex', with an appropriate representation for complex numbers.
   (b) Define a value, 'zero', which represents the complex number $0 + 0i$.
   (c) Define a value, 'c', which represents a complex number of the form $x + xi$.
   (d) Define functions, 'add' and 'mult', for addition and multiplication of complex numbers.
   (e) Define a function, 'f', which takes a complex number as argument and returns some complex number which is different.

4 Functions

1. Define a function, 'max', that returns the maximum of two integers in each of the following styles:
   (a) explicit function
   (b) implicit function
   (c) signature/axiom
2. Write a lambda expression which represents the same function as 'max'.
3. Write a lambda expression which represents the same function as 'twice' (p. 44 -45 in [RAI92]).
4. In [RAI92] there is a definition of a square root function, 'square_root'. Define a function, 'approx_sqrt', that for a given tolerance (positive real number) finds an approximation to the square root of non-negative real numbers. The approximation, approx_sqrt(x,eps), must be such that the mathematical square root, square_root(x), lies in the half open interval [approx_sqrt(x,eps),approx_sqrt(x,eps) + eps[.
5. Consider the specification, 'EQUIVALENCE_RELATION', p. 61 in [RAI92]. Give a similar specification but without use of axioms.
6. Consider the following function definition:

   value
   f : **Int** $\xrightarrow{\sim}$ **Int**
   f(x) ≡ f(x)

   Which functions (in particular which of the following functions) satisfy the definition?

   (a) **value**
   f : **Int** $\xrightarrow{\sim}$ **Int**
   f(x) ≡ **chaos**

   (b) **value**
   f : **Int** $\xrightarrow{\sim}$ **Int**
   f(x) ≡ 1

   (c) **value**
   f : **Int** → **Int**
   f(x) ≡ 1

7. Which functions satisfy:

   value
   f : **Int** → **Int**
   f(x) ≡ f(x)

8. Which functions satisfy:

   value
   f : **Int** $\xrightarrow{\sim}$ **Int**
   axiom
   ∀ x : **Int** • f(x) = f(x)

9. What is the difference in the function defined in 6(b) and 6(c)?
10. What is the type of: λ (x,y) : **Int** × **Int** • x ∗ y

5 Sets

1. Write a value expression representing the set of odd numbers between 0 and 10.
2. A university information system is to be specified. The system should keep track of the enrolled students and provided courses. For each course the system should keep track of the students attending that course. Define a module, 'UNIVERSITY_SYSTEM', containing definitions of
   (a) a type 'Student' of student identifications,
   (b) a type 'Course' of course identifications,
   (c) a type 'University' with an appropriate representation for universities, such that it for a given university is possible to extract information about students and courses as stated above,
   (d) a function , 'students', which gives the set of students of a given university,
   (e) a function , 'courses', which gives the set of courses of a given university,
   (f) a function, 'stud_of', which gives the set of students which are attending a given course at a given university,
   (g) a function, 'attending', which gives the set of courses which a given student is attending at a given university,
   (h) a function, 'new_stud', which enrols a new student at a university,
   (i) a function, 'drop_stud', which removes a student from a university,
   (j) a function, 'sizes_ok', which tests for a given university that there are no courses with more than 100 students or less than 5 students.

6 Lists

1. Define the following functions:
   (a) 'length' which calculates the length of a list, without using the built-in operator **len**,
   (b) 'rev', that reverses the elements of a list,
   (c) 'drev', that takes a list of lists of elements as argument, and doubly reverses it.
   Example:

   drev(⟨⟨e11, ..., e1i⟩,...,⟨en1, ..., enm⟩⟩) ≡
   ⟨⟨enm, ..., en1⟩,...,⟨e1i, ..., e11⟩⟩

2. Write an explicit definition of the function 'insert' that has an implicit definition in section 9.10 in [RAI92].

3. Write an explicit definition of the function 'sort' that has an implicit definition in section 9.9 in [RAI92].

4. Define a function, 'pascal', which generates Pascal triangles up to the order n. That is a list of n lists, the latter being the rows of the triangle. The first rows of such a triangle looks like:

   ⟨1⟩
   ⟨1 , 1⟩
   ⟨1 , 2 , 1⟩
   ⟨1 , 3 , 3 , 1⟩
   ⟨1 , 4 , 6 , 4 , 1⟩

   The nth row, for n>1, starts, and ends with 1, and its ith element, for 1<i<n, is the sum of the (i-1)st and ith element of the (n-1)st row.

5. A system for checking words on pages etc. is to be specified. A page consists of lines of words. Punctuation marks should be ignored. Define a module, 'PAGE', providing
   (a) types 'Page', 'Line' and 'Word',
   (b) a function, 'is_on', which checks that a given word is on a given page,
   (c) a function, 'number_of', which gives the number of occurrences of a particular word on a given page,
   (d) a type, 'Dict', of dictionaries of correctly spelled words,
   (e) a function, 'spell_check', that for a given page returns the words which are not correctly spelled with respect to a given dictionary of correct words.

7 Maps

1. Consider exercise 5.2. Propose another representation for universities and rewrite the specification accordingly.

8 Subtypes

1. Write subtype expressions for
   (a) finite integer lists of at least two elements
   (b) finite integer lists with no repetitions
   (c) finite non-empty sets of natural numbers
2. What are the maximal types of the types defined in previous question?
3. For a given type T, write a subtype expression which represents T-**set** as a subtype of T-**infset** and a subtype expression which represents T^* as a subtype of T^ω.
4. In each of the following cases decide what subtype relations there are between the T_is:
   (a) $T_1 = \{|\, i : \mathbf{Nat} \bullet \text{is\_even}(i) \,|\}$
   $T_2 = \{|\, i : \mathbf{Nat} \bullet \text{is\_even}(i) \land i \geq 5 \,|\}$
   (b) $T_1 = \{|\, i : \mathbf{Int} \bullet i/2 = 0 \,|\}$
   $T_2 = \{|\, i : \mathbf{Int} \bullet i/2 = 1 \,|\}$
   (c) $T_1 = A \to B$
   $T_2 = A \xrightarrow{\sim} B$
   (d) $T_1 = \mathbf{Int} \to \mathbf{Int}$
   $T_2 = \mathbf{Nat} \to \mathbf{Int}$
   $T_3 = \mathbf{Int} \to \mathbf{Nat}$
   $T_4 = \mathbf{Nat} \to \mathbf{Nat}$

9 Data Modelling

1. A hierarchical Unix-like file system is to be specified.

   A directory consists of named entities. An entity is a file or a directory. Files are not further specified. Names may be qualified.

   Define a modul, 'FILE_SYS', which provides appropriate types (e.g. 'Dir', 'Entity', 'File' and 'Name'), and the following functions:

   (a) 'wff_name_in_dir', which tests whether a name is well-formed wrt. a given directory,

   (b) 'look_up', which looks up a given name in a given directory,

   (c) 'ls', which gives the entities in a given directory.

2. A "bag" is an unordered collection of values of same type, possibly including duplicates.

   (a) Define a type 'Bag' of bags of elements of type 'Elem', where 'Elem' is some given type.

   (b) Define values 'count' (which for a given bag and a given element gives the number of occurrences of the element in the bag), 'empty', 'insert' and 'bag_union'.

3. A bank system is to be specified:

   Each customer may have zero, one or more accounts. An account has a balance. Each customer has an overdraft.

   Define a modul, 'BANK', which provides appropriate types, (e.g. a type 'Bank'), and the following functions:

   (a) 'new_customer', which creates a new customer in a given bank,

   (b) 'new_account', which creates a new account for a given customer in a given bank,

   (c) 'balance', which for a given account in a given bank gives the balance of the account,

   (d) 'delete_customer', which deletes a given customer in a given bank,

   (e) 'delete_account', which deletes a given account in a given bank.

4. Consider the specification of bills of products in section 10.8 in [RAI92].

   A more advanced system is to be specified. For each product it should also keep track of the number of occurrences of its subproducts.

   (a) Propose a new representation for 'Bop'.

   (b) Reformulate the specification accordingly.

   (c) Define a function, 'parts', which for a given product, p, and *bop*, in which it is recorded, yields a table which map each of p's subproducts to their total number of occurrences.

10 Variant Type Definitions

1. Consider the following specification:

   scheme
   SET =
   class
   type Set == empty | add(Elem, Set), Elem

   value
   is_in : Elem × Set → **Bool**

   axiom
   [is_in_empty] ∀ e : Elem • is_in(e, empty) ≡ **false**,

   [is_in_add] ∀ e, e1 : Elem, s : Set • is_in(e, add(e1, s)) ≡ e = e1 ∨ is_in(e, s),

   [unordered] ∀ e1, e2 : Elem, s : Set • add(e1, add(e2, s)) ≡ add(e2, add(e1, s)),

   [no_duplicates] ∀ e : Elem, s : Set • add(e, add(e, s)) ≡ add(e, s)
   end

   Is the following expression true in all models? (Justify your answer.)

   ∀ e1, e2, e3 : Elem • add(e1, add(e2, add(e3, empty))) ≡ add(e1, add(e3, add(e2, add(e1,empty))))

2. Expand the following variant type definitions into the definitions they are short-hands for:
   (a) **type** Figure == box(length : **Real**, width : **Real**) | circle(radius : **Real**)
   (b) **type** Figure == box(length : **Real**, width : **Real**) | circle(radius : **Real**) | _
   (c) **type** Figure == box(length : **Real**, width : **Real**) | circle(radius : **Real**) | _(base_line : **Real**)
   (d) **type** Tree == empty | node(left : Tree, val : Elem, right : Tree)

3. Consider the variant type definition of the type 'Tree'.
   Decide for each of the following expressions whether it is true in all models.
   (a) ∀ e : Elem, t1, t2 : Tree • t1 ≠ t2 ⇒ node(t1,e,t2) ≠ node(t2,e,t1)
   (b) ∀ e1, e2 : Elem, t1, t2 : Tree • e1 ≠ e2 ⇒ node(t1,e1,t2) ≠ node(t1,e2,t2)
   (c) ∀ e : Elem • node(empty,e,empty) ≠ node(node(empty,e,empty),e,empty)

4. Rewrite the specification of Peanos numbers (see section 7.14 in [RAI92]) such that a variant type definition is used.

5. Define a module 'ORIENTATION' which provides
   (a) a type 'Orientation' which has four values, one for each orientation (north, south, east and west)
   (b) functions 'turnleft', 'turnright' and 'opposite' which take an orientation and give a new orientation.

6. Prove

   $\forall$ v : Orientation • turnleft(v) $\neq$ v

7. Write an algebraic specification of stacks by defining a module 'STACK' which provides
   (a) types, 'Elem' and 'Stack',
   (b) a constant, 'empty', representing the empty stack,
   (c) a function, 'push', which pushes an element on a stack,
   (d) a function, 'top', which gives the top element (last inserted element) of a stack,
   (e) a function, 'pop', which removes the top element of a stack.

11 Case and Let Expressions

1. Define a function, 'f', that for $n \geq 0$ gives element number n in a sequence of numbers, a_i, determined by $a_0 = 0$, $a_1 = 1$ and $a_n = a_{n-2} + 2 * a_{n-1}$ for $n \geq 2$.

2. Consider the variant type definition of figures.

   type Figure == box(length : **Real**, width : **Real**) | circle(radius : **Real**)

   (a) Define a function, 'enlarge', that enlarges a figure with a factor 4.

   (b) What is enlarge(box(5.0,3.0))?

3. Define a function, 'rm3w', that takes a list of colours as argument and returns the same list except if it has a third element which is white, in which case this is removed from the list.

4. Define a function, 'dunion', that takes a set, ss, of sets of elements as argument and returns the set of all those elements which are elements of some set in ss.

   Example:

   dunion({ {1,2}, {7,1}, {5} }) = {1,2,7,5}

12 Union and Short Record Definitions

1. Specify a system to record information about vehicles.

   Vehicles are either cars or lorries. For cars as well as lorries the system should keep information about their current value and age. For lorries the system should additionally keep information about the tonnage.

   The system should also provide a function which for a given vehicle increases its age.

2. Rewrite the specification of vehicles, such that no union or short record type definitions are used.

13 Imperative Specification

1. Given a variable definition:

   variable x : **Int** := 0

   Reduce the following expressions:

   (a) x := 1; x := 2 ≡

   (b) x := x + 1 ≡

   (c) **initialise**; x ≡

   (d) ((x := 1 ; x) ≡ (x := 0 ; x+1)) ≡

   (e) ((x := 1 ; x) = (x := 0 ; x+1)) ≡

2. Write a model-oriented, imperative specification of stacks. The specification must provide a function 'empty' which makes the stack empty, a function 'push' which pushes an element on the stack, a function 'is_empty' which tests whether the stack is empty, a function 'top' which gives the top element (last inserted element) of the stack (without removing it) and a function 'pop' which removes the top element of the stack.

   Use explicit function definitions.

3. Write a model-oriented, imperative specification of stacks. Use implicit function definitions (i.e. use post expressions – don't use equivalence expressions).

4. Write an algebraic, imperative specification of stacks.

5. Consider the model-oriented, imperative specification of databases in section 18.11 in [RAI92]. Write a corresponding specification using post expressions instead of equivalence expressions.

14 Concurrency

1. Define a process that continuously inputs two integers from two channels *l1* and *l2*, one from each channel, and outputs their maximum on a channel *r*.
2. Define a process that continuously inputs two integers from two channels *l1* and *l2*, one from each channel, and outputs their maximum on one channel *r1* and their minimum on another channel *r2*.
3. Concrete concurrent stack:

   Define a module, 'CC_STACK', that provides

   (a) A type, 'Elem', of elements, and a type, 'Stack', of stacks of elements. 'Stack' should be given some concrete representation in terms of 'Elem'.

   (b) A stack process (function), 'stack_p', which is defined explicitly as an external choice between four communication behaviours (one for emptying the stack, one for pushing an element to the stack, one for testing whether the stack is empty, and one for poping an element off the stack). The stack process should take a stack as argument.

   (c) Channels.

   (d) Functions 'empty', 'push', 'is_empty' and 'pop', which serve as interface to the 'stack_p' process.
4. Concrete concurrent imperative stack:

   How could you change 'CC_STACK' such that 'stack_p' does not take a stack as argument?
5. Abstract concurrent stack:

   Define a module, 'AC_STACK', which is more abstract than 'CC_STACK' in that

   (a) No concrete representation for stacks is given.

   (b) No channels are defined.

   (c) The stack process and interface functions are not explicitly defined, instead axioms relating parallel composition of the stack process and the interface functions are given.
6. Consider the following specification:

```
scheme SEMAPHORE =
  hide get, release, semaphore in
  class
    type
      Process = Unit ~→ in any out any Unit
   channel get, release : Unit
    value
      system, p1, p2, p3, f1, f2, f3, semaphore : Process
    axiom
      p1() ≡ f1() ; p1(),
      p2() ≡ f2() ; p2(),
      p3() ≡ f3() ; p3(),
      semaphore() ≡ skip,
      system() ≡ p1() || p2() || p3() || semaphore()
  end
```

   We want to use the semaphore to ensure that only one of the processes 'f1', 'f2', and 'f3' can run at any one time, regardless of how they are implemented. Change the first 4 axioms only of 'SEMAPHORE' to achieve this.

   Hint: The accesses of 'p1', 'p2', 'p3' and 'semaphore' mean that they can input from or output to the channels 'get' and 'release'.

15 Modularity

1. Stack development.

   (a) Write an algebraic applicative specification of stacks by defining a scheme, 'PSTACK1', which provides:

   i. a type 'Stack',
   ii. a constant, 'empty', representing the empty stack,
   iii. a function, 'push', which pushes an element on a stack,
   iv. a function, 'top', which gives the top element (last inserted element) of a stack,
   v. a function, 'pop', which removes the top element of a stack.
   vi. a function, 'is_empty', which tests whether a stack is empty.

   Hint 1: Be sure that you only get models which have the desired properties, e.g. that all stacks must be finitely generated by the constructors, 'empty' and 'push'.

   Hint 2: Use parameterization.

   (b) Define a scheme, 'INT_STACK', for stacks of integers.

   (c) Propose a development of 'PSTACK1' to a more concrete scheme, 'PSTACK2'.

2. Define a scheme, 'STACK_STACK', for stacks of stacks of elements. The type of stacks of stacks of elements should have the name 'Stack_of_stacks'.

   Expand your definition to an equivalent definition where the right-hand side is a basic class expression.

3. Consider the specification, 'CC_STACK', defined in exercise 14.3.

   Propose a more general specification, 'PCC_STACK', by using parameterization. Can you reuse 'PSTACK2' in this specification?

4. Consider the object, 'SYSTEM', defined p. 239 in [RAI92].

   Propose a more general specification (a parameterized scheme), 'SYSTEM_S'.

   How can you obtain 'SYSTEM' from 'SYSTEM_S'?

5. A circular buffer is to be specified in an explicit, imperative style.

   A circular buffer has a maximal size, 'max'. The circular buffer should contain as many variables, v_1, ..., v_{max}, as 'max' indicates, and some variables, 'start' and 'next', keeping track of where the buffer starts and ends. In the initial state 'start' and 'next' should be 1.

   Example:

   If 'max' = 5 and the contents of 'start' and 'next' are 2 and 4, respectively, then the buffer contain two elements (in v_2 and v_3) and a put(e) would assign v_4 to e and 'next' to 5, while a get would assign 'start' to 3 and return the contents of v_2.

   Define a scheme 'CIRCULAR_BUFFER', that is parameterized wrt. the kind of elements in the buffer and the size, 'max', of the buffer. The scheme should provide to the surroundings:

   (a) a function, 'put', that puts an element into the buffer

   (b) a function, 'get', removes an element from the buffer and returns that element as result

   Hint: Define two auxilliary functions 'is_empty' and 'is_full' that test whether the buffer is empty and full, repectively.

16 Refinements

1. Which of the following type definitions are implementations (refinements) of each other?
   (a) **type** t
   (b) **type** t = **Int**
   (c) **type** t = **Nat**
2. Which of the following value definitions are implementations (refinements) of each other?
   (a) **value** x : **Int**
   (b) **value** x : **Int** • x > 2
   (c) **value** x : **Int** = 2
   (d) **value** x : **Nat**
   (e) **value** x : **Nat** • x > 2
   (f) **value** x : **Nat** = 2
   (g) **value** x : **Nat** • x < 0
3. Which of the following function definitions are implementations (refinements) of each other?
   (a) **value**
   f : **Int** $\xrightarrow{\sim}$ **Int**
   f(x) ≡ x
   pre x ≥ 0
   (b) **value**
   f : **Int** $\xrightarrow{\sim}$ **Int**
   f(x) ≡ x
   pre x ≥ 2
   (c) **value**
   f : **Int** → **Int**
   f(x) ≡ x
4. Consider the development of 'PSTACK1' to 'PSTACK2' in exercise 15.1.
   (a) Does 'PSTACK2' statically implement 'PSTACK1'?
   (b) What are the refinement conditions for the development? (Write them in the form of an axiom declaration.)

17 Verification

Consider the following development relation and specifications:

devt_relation STACK_DEV (STACK2 **for** STACK1) :
$\vdash$ STACK2 $\preceq$ STACK1

scheme
STACK1 =
class
type Elem, Stack == empty | push(top : Elem, pop : Stack)

value
is_empty : Stack $\to$ **Bool**
is_empty(st) $\equiv$ st = empty
end

scheme
STACK2 =
class
type Elem, Stack = Elem*

value
empty : Stack = $\langle\rangle$,

push : Elem $\times$ Stack $\to$ Stack
push(e, st) $\equiv$ $\langle$e$\rangle$ $\frown$ st,

top : Stack $\xrightarrow{\sim}$ Elem
top(st) $\equiv$ **hd** st **pre** st $\neq$ $\langle\rangle$,

pop : Stack $\xrightarrow{\sim}$ Stack
pop(st) $\equiv$ **tl** st **pre** st $\neq$ $\langle\rangle$,

is_empty : Stack $\to$ **Bool**
is_empty(st) $\equiv$ st = $\langle\rangle$
end

1. Does STACK2 statically implement STACK1?
2. What are the implementation conditions for the development?
3. Justify the development conditions except the induction axiom by using the proof rules for unfolding of function applications and value names, and the proof rules below.

   The justification should be formal with the following two exceptions:

   (a) You need not to expose readonly and convergent applicability conditions and their justification in your justifications if you can convince your-self that they are obviously satisfied.

   (b) Other applicability conditions should be exposed and their justification may be informal.

assume type T **value** eb : **Bool in**
inference_rules
[all_assumption_inf]
id : T $\vdash$ eb
$---------$
$\forall$ id : T • eb

assume type T **value** e, e′ : T, el : T^{ω}, eb : **Bool in**
value_rules

[equality_annihilation]
e = e $\simeq$ **true** **when readonly**(e) $\wedge$ **convergent**(e)

[is_annihilation]
e $\equiv$ e $\simeq$ **true**

[hd_concatenation2]
hd($\langle$e$\rangle^\frown$el) $\simeq$ e **when readonly**(el) $\wedge$ **convergent**(el)

[tl_concatenation2]
tl($\langle$e$\rangle^\frown$el) $\simeq$ el **when readonly**(e) $\wedge$ **convergent**(e)

[empty_list_inequality2]
$\langle\rangle \neq \langle$e$\rangle^\frown$el $\simeq$ **true**
when convergent(e) $\wedge$ **readonly**(e) $\wedge$ **convergent**(el) $\wedge$ **readonly**(el)

[empty_list_inequality3]
$\langle$e$\rangle^\frown$el $\neq \langle\rangle \simeq$ **true**
when convergent(e) $\wedge$ **readonly**(e) $\wedge$ **convergent**(el) $\wedge$ **readonly**(el)

[not_true]
$\sim$**true** $\simeq$ **false**

[not_not]
$\sim \sim$ eb $\simeq$ eb

[in_equality_expansion]
e $\neq$ e′ $\simeq \sim$ (e = e′)

18 Specification of Language-based Systems

1. A simple typed applicative language, SAL3, of expressions is to be specified.

   A SAL3 *expression* is either a constant, an infix expression, a prefix expression, an identifier or a let expression.
   A *constant* is either an integer constant or a boolean constant.
   An *infix expression* consists of an infix operator and two operand expressions.
   A *prefix expression* consists of a prefix operator and an operand expression.
   A *let expression* consists of an identifier and two expressions.

   There are *infix operators* for integer addition, integer subtraction, integer equality, Boolean conjunction, Boolean disjunction and Boolean equality.
   There are *prefix operators* for integer negation and Boolean negation.

   Concrete syntax examples:

   5
   true
   $2 + 4$
   true $\wedge$ $(5 = 8)$
   -1
   let x $=$ 7 **in** x $+$ **3** $-$ 1 **end**

   (a) Define an abstract syntax for SAL3, i.e. define a type of SAL3 expressions.
   (b) Define an appropriate static semantics for SAL3, i.e. define a function which tests whether a SAL3 expression is well-formed.
   Hint: Define an auxilliary function, 'type_of', which extracts the type of a SAL3 expression.
   (c) Define a dynamic semantics for SAL3, i.e. a function which gives meaning to (statically correct) SAL3 expressions.

2. A simple imperative language, SIL3, of programs is to be specified.

   A SIL3 *program* consists of zero, one or more variable declarations followed by a statement.

   A *variable declaration* consists of a name of a variable and a constant (natural number), the latter representing the initial contents of the variable.

   A *statement* may be an assignment, an if statement, a for statement or a sequence of two statements.
   An *assignment* consists of the name of a variable and an expression.
   An *if statement* consists of an expression and two statements. Here expressions evaluating to zero should be interpreted as "true".
   A *for statement* consists of a name, a (from) expression, a (to) expression and a statement in which the name may be used.

   An *expression* is either a constant (a natural number), a plus expression or a name (introduced in a variable declaration or in an enclosing for statement). A *plus expression* consists of two expression.

   Concrete syntax examples:

   variable
   x $= 2$,
   y $= 7$
   begin
   x $:=$ y $+ 1$;
   for z $= 1$ **to** 20 **do**
   x $:=$ x $+$ z
   end;
   if x **then** y $:= 5$ **else** y $:= 3$ **end**
   end

(a) Define an abstract syntax for SIL3, i.e. define a type of programs.

(b) Define an appropriate static semantics for SIL3, i.e. define a function which tests whether a program is well-formed.

(c) Define a dynamic (applicative) semantics for SIL3, i.e. a function which gives meaning to (statically correct) programs.

3. A language of differentiable expressions and a function which perform derivations of such expressions is to be specified.

   There are the following kinds of expressions (explained by concrete syntax examples):

   - constants (1.0)
   - variables (x)
   - addition expressions (e1 + e2)
   - multiplication expressions (e1 ∗ e2)
   - monadic minus expressions (− e)
   - sine expressions (**sin** e)
   - cosine expressions (**cos** e)

   (a) Define an abstract syntax for the language of differentiable expression.

   (b) For the expression: 1.0 + **cos** x + **sin** (2.0∗x) give a corresponding abstract expression.

   (c) Define a function, 'diff', which applies to a differentiable expression and produces its derivative wrt. a given variable.

4. Abstract syntax and semantics of a one-sorted algebraic specification language is going to be specified. The sentences in the language are 1. order predicate calculus formulas.

   Informal description of the syntax:

   A *specification* consists of a signature and 0, 1 or more sentences.

   A *signature* consists of declarations of functions and predicates. A declaration of a function consists of a function name and an arity. A declaration of a predicate consists of a predicate name and an arity.

   A *sentence* is a formula (with no free variables).

   A *formula* can be an application of a predicate name to a (possibly empty) list of terms, a negation of a formula, an implication between two formulas, or a universal quantification of a formula wrt. a variable.

   A *term* is a variable name or an application of a function name to a (possibly empty) list of terms.

   (a) Define an abstract syntax for specifications.

   (b) Define a static semantics for specifications.

   Hint 0: The sentences must conform to the signature.

   Hint 1: Define a type, 'Dict', of dictionaries and define an auxiliary function, 'dict', which for a given signature gives a dictionary. Dictionaries may be seen as a more abstract representation of signatures.

   (c) We are now going to define the dynamic semantics of specifications.

   Informally, the mening of a specification is the collection of all interpretations that each (conforms to the signature and) satisfies each of the sentences.

   An *interpretation* binds function names to function values and predicate names to predicates values.

   A sentence *satisfies an interpretation* if ...

   An *assignment* binds variable names to values.

   We define the dynamic semantics in a number of steps:

   i. Define an abstract type, 'Val', of denotations of variables. Define types 'FunVal' and 'PredVal' of function values and predicate value, respectively, in terms of 'Val'.

   ii. Define a type, 'Interpretation', of interpretations.

   iii. Define a function that tests whether an interpretation conforms to a dictionary, i.e. gives denotations to the names in the dictionary, and that the denotations are right wrt. the arities.

   iv. Define a type, 'Asg', of assignments.

   v. Define a function, 'eval', that evaluates a term in an interpretation wrt. a given assignment.

   vi. Define a function, 'eval', that evaluates a formula in an interpretation wrt. a given assignment.

   vii. Define an function, 'satisfied', that evaluates a sentence in an interpretation.

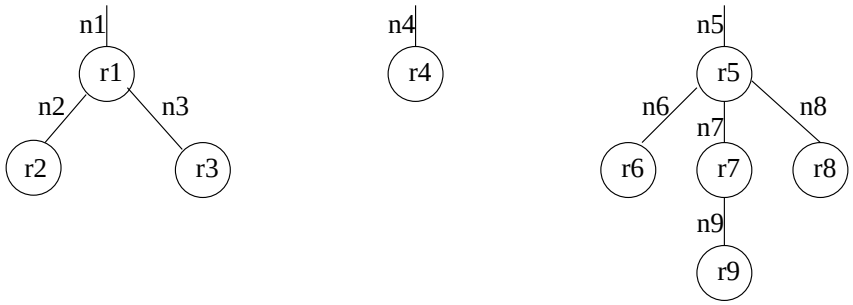
   viii. Define a meaning function for specifications.

19 Sample Exam Questions

1. Data modelling

   A forest consists of a finite collection of uniquely named trees. A tree consists of a root and a forest. Tree names and roots are not further specified.

   Example of a forest:



   (a) Define a type, 'Forest', of forests and a type, 'Tree', of trees.

   (b) Assume given the following value definitions:

```
value
  n1, n2, n3, n4, n5, n6, n7, n8, n9 : Name,
  r1, r2, r3, r4, r5, r6, r7, r8, r9 : Root,
```

   Define a value, 'a_forest', that represents the forest in the figure.

   (c) Define a function 'only_ones' which tests for a given forest that tree names must only appear ones.

2. Data modelling

   A marriage bureau is to be specified. The mariage bureau has a database containing informations about which female clients want to be married, which male clients want to be married, and which couples are married.

   Define a module 'MARRIAGE_BUREAU', which provides:

   (a) types, 'Woman', 'Man' and 'Database',

   (b) a function, 'is_married', which tests whether a woman is registrated as married, and a similar function for men,

   (c) functions, 'register_woman' and 'register_man', which are used to register new clients in a database,

   (d) a function, 'marry', which is used to update a database when two clients are married with each other.

   The specification style should be applicative and explicit.

   Hint 1: Remember possible well-formedness conditions on databases.

   Hint 2: Remember appropriate pre-conditions for the functions.

3. Algebraic specification

   Write an algebraic applicative specification of finite, deterministic maps by defining a module, 'MAP', which provides:

   (a) a type 'Map',

   (b) a constant 'empty' representing the empty map,

   (c) a function 'add' , which adds a domain element and a range element to a map,

   (d) functions **'eq'**, **'dom'**, **'rng'**, '\', '†' and 'apply' with the obvious meanings.

   Be sure that the specification only has models with desired properties.

4. Imperative specification

   Consider the following applicative specification, 'POOL', of a pool:

```
scheme
  POOL(E1 : ELEM, E2 : ELEM) =
    class
      type Pool = E1.Elem m→ E2.Elem-set

      value
        init : Pool = [ ],

        exist : E1.Elem × Pool → E2.Elem-set
        exist(e1, p) ≡ if e1 ∈ dom p then p(e1) else {} end,

        create : E1.Elem-set × E2.Elem × Pool → Pool
        create(e1s, e2, p) ≡ p † [ e1 ↦ exist(e1, p) ∪ {e2} | e1 : E1.Elem • e1 ∈ e1s ],

        destroy : E1.Elem-set × E2.Elem × Pool → Pool
        destroy(e1s, e2, p) ≡ p † [ e1 ↦ exist(e1, p) \ {e2} | e1 : E1.Elem • e1 ∈ e1s ]
    end
```

   Write a similar but imperative specification of a pool. The specification should be explicit wrt. variables and all functions should be explicitly defined.

5. Process specification

   Write an concurrent specification of a pool.

   There should be a pool process, 'pool_p', and interface functions (processes). The specification should be explicit wrt. channels and all functions should be explicitly defined.

6. Modularity

   Expand the definition of the 'VISITOR_POOL' scheme:

```
scheme
  VISITOR_POOL(D : DATE, V : VISITOR) =
    use visitors for exist, create_visitor for create, destroy_visitor for destroy in
      POOL(D{Date for Elem}, V{Visitor for Elem})
```

where 'POOL' is the applicative scheme shown in exercise 4, and 'DATE' and 'VISITOR' are as shown below:

scheme
DATE = **class type** Date **end**

scheme
VISITOR = **class type** Visitor **end**

7. Refinements

   Which of the following variable definitions are implementations (refinements) of each other?

   (a) **variable** v : **Int** := 2

   (b) **variable** v : **Int** := 1 + 1

   (c) **variable** v : **Int**
   axiom initialise ; v := 2 ≡ **initialise**

   (d) **variable** v : **Int**

8. Language description

   Abstract syntax and static semantics for an imperative loop-exit language is to be specified.

   A *program* consists of a sequence of declarations, followed by a sequences of statements. A *declaration* consists of a name of a variable. A *statement* is either an assignment, a loop, or an exit. An *assignment* consists of a variable name and an expression. A *loop* is a sequence of sentences. An *exit* consists of an expression. Expressions are not further specified.

   (a) Define an abstract syntax for the language of programs (i.e. define types of programs, statements, etc.)

   (b) Define an appropriate static semantics for the language (i.e. define appropriate well-formedness functions). Formulate in words which context conditions you have imposed on the language.

References

[Bjø] D. Bjørner. Software Architechtures and Programming Systems Design — the VDM Approach, 2 vols. To be published.

[GP92] C. George and S. Prehn. The RAISE Justification Handbook. LACOS Report DOC/7, Computer Resources International A/S, May 1992.

[Hax94] A. Haxthausen. Formal Software Specification Using RAISE — Exercises. Technical report, Department of Computer Science, Technical University of Denmark, April 1994.

[HP92] A.E. Haxthausen and J. Storbank Pedersen. Course Guide for EuroPACE Video Course. LACOS Report DOC/11, Computer Resources International A/S, January 1992.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall Series in Computer Science. Prentice-Hall International, 1986.

[RAI92] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioner Series. Prentice-Hall International, 1992.