# An RSL Subset (V1)

Anne E. Haxthausen

DTU Compute, February 14, 2024

# Contents

|  |  |  |
| --- | --- | --- |
| **1** | **Syntax for RSL Subset V1** | **1** |
|  | Scheme Declarations | 3 |
|  | Class Expressions | 3 |
|  | Declarations | 3 |
|  | Type Expressions | 3 |
|  | Value Expressions | 4 |
|  | Typings | 4 |
|  | Names | 4 |
|  | Operators and Connectives | 4 |
| **2** | **Precedence and Associativity of Operators for Full RSL** | **5** |
| **3** | **Lexical Matters for Full RSL** | **6** |
|  | Varying Tokens | 6 |
|  | ASCII Forms of Greek Letters | 8 |
|  | Fixed Tokens | 8 |
|  | RSL Keywords | 9 |
| **A** | **About Syntax and Context Rules** | **10** |
|  | Syntax Conventions | 10 |
|  | Static Correctness | 12 |

2

CHAPTER 1

# Syntax for RSL Subset V1

This section presents a grammar for a subset of RSL. The subset excludes the following constructs from full RSL:

1. module structuring constructs, i.e.:
   1. object delarations are excluded,
   2. a specification now consists of only *one* scheme definition,
   3. the scheme definition must not have formal parameters,
   4. its constituent class expression must be a basic class expression - all other kinds of class expressions are disallowed, and
   5. a basic class expression can now only contain type, value, and axiom declarations (not variable, channel, and object declarations)
2. multiple typings (only single typings are allowed)
3. product bindings (so only id is allowed as a binding)
4. implicit value definitions
5. implicit function definitions (using pre/post specs)
6. user-defined operators
7. imperative constructs (e.g. variable declarations, assignments and loops)
8. process/concurrency constructs, (e.g. channel declarations, input and output expressions, and concurrency combinators)
9. types and associated value expressions and operators for natural numbers, reals, texts (strings), chars, the unit value, products (tuples), functions, sets, lists, maps (tables), union, records, and record variants (only variants corre- sponding to enumerations are allowed)
10. case expressions
11. elsif branches inside if-then-else constructs
12. local expressions
13. equivalence expressions
14. post expressions
15. higher-order functions and higher-order function application expressions, e.g. application expr can only be on the form id(value expr-list)

1

### Changes to the grammar

The grammar for the subset of RSL presented in this chapter has been achieved from the grammar of the full RSL language by removing many alternatives and sometimes making optional constructs either non-optional or totally removing them. Furthermore, in some cases, in the right-hand side of a rule *t ::= ... t1 ...*, a non- terminal *t1* has been replaced with the right-hand side *rhs* of the grammar rule *t1*

*::= rhs* for *t1* (when the right-hand side *rhs* only had one alternative) and the rule for *t1* was removed. So the first rule became *t ::= ... rhs ...*.

As an example, the original grammar of bindings was:

binding ::= id *|*

id product binding

product binding ::= ( binding-list2 )

Then first *product binding* was removed from the language, resulting in the gram- mar:

binding ::= id

Then all appearences of *binding* in right-hand sides of grammar rules were replaced with *id* and the rule for *binding* was removed.

The first kind of grammar changes was necessary for reducing to the sub-language. Later on, it would be easy to re-introduce the removed constructs, e.g. by adding an alternative. The second kind of grammar changes were not necessary, but only made in order to simplify the grammar and make parse trees smaller. A disad- vantage of the latter kind of grammar changes is that it becomes less easy to re-introduce removed constructs. However, it was judged that reducing the size of parse trees was more important as we had examples where large parse trees were an issue.

### Suggested changes to static semantics

The static semantics is restricted by:

1. disallowing overloading
2. disallowing recursive functions
3. maybe enforcing the define-before-use paradigm and later remove this restric- tion???

## Scheme Declarations

scheme decl ::=

**scheme** scheme def

explicit value def ::=

id : type expr = *pure-*value expr

explicit function def ::=

id : type expr

*~* type expr

scheme def ::=

id = class expr

## Class Expressions

class expr ::=

**class** opt-decl-string **end**

## Declarations

decl ::=

type decl *|*

value decl *|*

axiom decl

### Type Declarations

type decl ::=

**type** type def-list type def ::=

sort def *|*

variant def *|* sort def ::= abbreviation def

id

variant def ::=

id == id-choice abbreviation def ::= id = type expr

### Value Declarations

value decl ::=

**value** value def-list value def ::=

value signature *|*

explicit value def *|*

explicit function def

value signature ::= id : type expr

-product *→*

id ( id-list ) *≡* value expr

### Variable Declarations

variable decl ::=

**variable** variable def-list variable def ::=

id : type expr opt-initialisation

initialisation ::=

:= *pure-*value expr

NOTE non-terminal variable decl is not used in any grammar rule for the defined RSL subset, but in the grammar rules for the RSL\* exten- sion. Since variable decl stems from full RSL, it is placed here.

### Axiom Declarations

axiom decl ::=

**axiom** axiom def-list axiom def ::=

*readonly logical-*value expr

axiom naming ::=

[ id ]

## Type Expressions

type expr ::=

type literal *|*

*type-*name *|* subtype expr *|* bracketed type expr

type literal ::=

#### Bool *|*

**Int**

subtype expr ::=

*{|* single typing *• pure logical-*value expr *|}*

bracketed type expr ::= ( type expr )

## Value Expressions

value expr ::=

value literal *|*

*value or variable-*name *|* application expr *|* quantified expr *|* bracketed expr *|*

axiom infix expr *|* value infix expr *|* axiom prefix expr *|* value prefix expr *|* let expr *|*

if expr

value literal ::=

bool literal *|*

bool literal ::= int literal

#### true *|*

**false**

application expr ::=

*function value-*id ( value expr-list )

quantified expr ::=

quantifier single typing-list *•*

*pure logical-*value expr

quantifier ::=

*∀ |*

*∃ |*

*∃*!

bracketed expr ::= ( value expr ) axiom infix expr ::=

*logical-*value expr infix connective

value infix expr ::= *logical-*value expr

value expr infix op value expr

axiom prefix expr ::=

prefix connective *logical-*value expr

value prefix expr ::=

prefix op value expr

let expr ::=

**let** id = value expr **in** value expr **end**

if expr ::=

**if** *logical-*value expr

**then** value expr **else** value expr **end**

## Typings

single typing ::=

id : type expr

## Names

name ::= id

## Operators and Connectives

infix op ::=

= *|*

*̸*= *|*

*> |*

*< |*

*≥ |*

*≤ |*

+ *|*

*— |*

*∗ |*

/

prefix op ::=

#### abs

infix connective ::=

*⇒ |*

*∨ |*

*∧*

prefix connective ::=

*~*

CHAPTER 2

# Precedence and Associativity of

**Operators for Full RSL**

This section is taken from the RSL book.

|  |  |  |
| --- | --- | --- |
| Value operator precedence – increasing | | |
| Prec | Operator(s) | Associativity |
| 14 | *2 λ ∀ ∃ ∃*! | Right |
| 13 | *≡* **post** |  |
| 12 | *⌊⌈⌉⌋ ⌈⌉*  – | Right |
| 11 | ; | Right |
| 10 | := |  |
| 9 | *⇒* | Right |
| 8 | *∨* | Right |
| 7 | *∧* | Right |
| 6 | = *̸*= *> < ≥ ≤ ⊂ ⊆ ⊃ ⊇ ∈ ̸∈* |  |
| 5 | + *− \* ^ *∪ †* | Left |
| 4 | *∗ / ◦ ∩* | Left |
| 3 | *↑* |  |
| 2 | : |  |
| 1 | *~* |  |

|  |  |  |
| --- | --- | --- |
| Type operator precedence – increasing | | |
| Prec | Operator(s) | Associativity |
| 3 | *~*  *→m → →* | Right |
| 2 | *×* |  |
| 1 | **-set -infset** *∗ ω* |  |

5

CHAPTER 3

# Lexical Matters for Full RSL

This chapter describes lexical matters, i.e. the microsyntax for RSL.

Basically, RSL follows the rules now in current practice for most programming languages: a text (i.e. an RSL specification) is represented as a string of characters, which is interpreted left-to- right and broken into a string of tokens. The characters are drawn from a superset of the ASCII characters called the *full RSL character set* . Tokens may be separated by ‘whitespace’, which is strings of one or more of the following characters: line-feed, carriage-return, space and tab. (Note that *comments* are part of the RSL syntax and thus cannot be used freely as whitespace. Also note that comments may not be nested.)

There are two types of tokens in RSL: varying and fixed.

## Varying Tokens

The microsyntax for varying tokens is defined by the syntax rules below, where the characters used in forming tokens are shown in quotes, as in ‘$’. Furthermore, LF, CR and TAB are used to denote the ASCII characters line-feed, carriage-return and tab.

id ::=

letter opt-letter or digit or underline or prime-string letter or digit or underline or prime ::=

letter *|* digit *|* underline *|* prime

letter ::=

ascii letter *|* greek letter

comment ::=

‘/’ ‘*∗*’ comment item-string ‘*∗*’ ‘/’

comment item ::= comment char

comment char ::=

LF *|* CR *|* TAB *|* ascii letter *|* digit *|* graphic *|* prime *|* quote *|* backslash

int literal ::=

6

digit-string

real literal ::=

digit-string ‘.’ digit-string text literal ::=

‘*′′*’ opt-text character-string ‘*′′*’

char literal ::=

‘*′*’ char character ‘*′*’

text character ::= character *|* prime

char character ::= character *|* quote

character ::=

ascii letter *|* digit *|* graphic *|* escape

digit ::=

‘0’ *|* ‘1’ *|* ‘2’ *|* ‘3’ *|* ‘4’ *|* ‘5’ *|* ‘6’ *|* ‘7’ *|* ‘8’ *|* ‘9’

ascii letter ::=

‘a’ *|* ‘b’ *|* ‘c’ *|* ‘d’ *|* ‘e’ *|* ‘f’ *|* ‘g’ *|* ‘h’ *|* ‘i’ *|* ‘j’ *|* ‘k’ *|* ‘l’ *|* ‘m’ *|*

‘n’ *|* ‘o’ *|* ‘p’ *|* ‘q’ *|* ‘r’ *|* ‘s’ *|* ‘t’ *|* ‘u’ *|* ‘v’ *|* ‘w’ *|* ‘x’ *|* ‘y’ *|* ‘z’ *|*

‘A’ *|* ‘B’ *|* ‘C’ *|* ‘D’ *|* ‘E’ *|* ‘F’ *|* ‘G’ *|* ‘H’ *|* ‘I’ *|* ‘J’ *|* ‘K’ *|* ‘L’ *|* ‘M’ *|*

‘N’ *|* ‘O’ *|* ‘P’ *|* ‘Q’ *|* ‘R’ *|* ‘S’ *|* ‘T’ *|* ‘U’ *|* ‘V’ *|* ‘W’ *|* ‘X’ *|* ‘Y’ *|* ‘Z’

greek letter ::=

‘`alpha’ *|* ‘`beta’ *|* ‘`gamma’ *|* ‘`delta’ *|* ‘`epsilon’ *|* ‘`zeta’ *|* ‘`eta’ *|* ‘`theta’ *|* ‘`iota’ *|* ‘`kappa’ *|* ‘`mu’ *|* ‘`nu’ *|* ‘`xi’ *|* ‘`pi’ *|* ‘`rho’ *|* ‘`sigma’ *|* ‘`tau’ *|* ‘`upsilon’ *|* ‘`phi’ *|* ‘`chi’ *|* ‘`psi’ *|* ‘`omega’ *|* ‘`Gamma’ *|* ‘`Delta’ *|* ‘`Theta’ *|* ‘`Lambda’ *|* ‘`Xi’ *|* ‘`Pi’ *|* ‘`Sigma’ *|* ‘`Upsilon’ *|* ‘`Phi’ *|* ‘`Psi’ *|* ‘`Omega’

underline ::= ‘’

prime ::= ‘*′*’

quote ::= ‘*′′*’

backslash ::= ‘*\*’

graphic ::=

‘ ’ *|* ‘!’ *|* ‘*#*’ *|* ‘$’ *|* ‘%’ *|* ‘&’ *|* ‘(’ *|* ‘)’ *|* ‘*∗*’ *|* ‘+’ *|* ‘,’ *|* ‘*−*’ *|* ‘.’ *|* ‘/’ *|* ‘:’ *|* ‘;’ *|*

‘*<*’ *|* ‘=’ *|* ‘*>*’ *|* ‘?’ *|* ‘@’ *|* ‘[’ *|* ‘]’ *|* ‘^’ *|* ‘’ *|* ‘`’ *|* ‘*{*’ *|* ‘*|*’ *|* ‘*}*’ *|* ‘*∼*’

escape ::=

‘*\*’‘r’ *|* ‘*\*’‘n’ *|* ‘*\*’‘t’ *|* ‘*\*’‘a’ *|* ‘*\*’‘b’ *|* ‘*\*’‘f’ *|* ‘*\*’‘v’ *|* ‘*\*’‘?’ *|*

‘*\*’‘*\*’ *|* ‘*\*’‘*′*’ *|* ‘*\*’‘*′′*’ *|* ‘*\*’ oct constant *|* ‘*\*’‘x’ hex constant

oct constant ::= oct digit *|* oct digit oct digit *|* oct digit oct digit oct digit hex constant ::=

hex digit-string

oct digit ::=

‘0’ *|* ‘1’ *|* ‘2’ *|* ‘3’ *|* ‘4’ *|* ‘5’ *|* ‘6’ *|* ‘7’

hex digit ::=

digit *|* ‘a’ *|* ‘b’ *|* ‘c’ *|* ‘d’ *|* ‘e’ *|* ‘f’ *|* ‘A’ *|* ‘B’ *|* ‘C’ *|* ‘D’ *|* ‘E’ *|* ‘F’

## ASCII Forms of Greek Letters

Greek letters, which may be used in identifiers, have ASCII forms as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| ASCII | Full | ASCII | Full |
| `alpha  `beta  `gamma  `delta  `epsilon  `zeta  `eta  `theta  `iota  `kappa  `mu  `nu  `xi  `pi  `rho  `sigma  `tau  `upsilon  `phi  `chi  `psi  `omega | *α β γ δ ϵ ζ η θ ι κ*  *µ ν ξ π ρ σ τ υ ϕ χ ψ ω* | `Gamma  `Delta  `Theta  `Lambda  `Xi  `Pi  `Sigma  `Upsilon  `Phi  `Psi  `Omega | Γ  ∆  Θ Λ  Ξ Π Σ  Υ Φ Ψ  Ω |

## Fixed Tokens

The representation of individual fixed tokens is given directly in the syntax rules for RSL. However, a representation using only ASCII characters is possible, as defined in the following table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ASCII | Full | ASCII | Full | ASCII | Full |
| >< | *×*    *⌊⌈⌉⌋*  *↑*  *∧*  *≥*  *≤*  *∩*  *→m*  *~*  *→*  *→*  *◦*  *•* | isin | *∈* | ~isin | *̸∈* |
| || | ++ | – | -\ | *λ* |
| |=| | |^| | *⌈⌉* | -list | *∗* |
| \*\* | -inflist | *ω* | ~= | *̸*= |
| /\ | \/ | *∨* | +> | *'→* |
| >= | exists | *∃* | all | *∀* |
| <= | union | *∪* | !! | *†* |
| inter | << | *⊂* | always | *2* |
| -m-> | <<= | *⊆* | => | *⇒* |
| -~-> | >> | *⊃* | is | *≡* |
| -> | >>= | *⊇* | <-> | *↔* |
| # | <. | *⟨* | .> | *⟩* |
| :- |  |  |  |  |

The word equivalents of certain symbols: all, exists, union, inter, isin, always are re- served, and cannot be used as identifiers.

## RSL Keywords

The RSL keywords are listed below. They cannot be used as identifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Keywords for RSL | | | |
| **Bool** | **class** | **inds** | **skip** |
| **Char** | **do** | **initialise** | **stop** |
| **Int** | **dom** | **int** | **swap** |
| **Nat** | **elems** | **len** | **then** |
| **Real** | **else** | **let** | **tl** |
| **Text** | **elsif** | **local** | **true** |
| **Unit** | **end** | **object** | **type** |
| **abs** | **extend** | **of** | **until** |
| **any** | **false** | **out** | **use** |
| **as** | **for** | **post** | **value** |
| **axiom** | **forall** | **pre** | **variable** |
| **card** | **hd** | **read** | **while** |
| **case** | **hide** | **real** | **with** |
| **channel** | **if** | **rng** | **write** |
| **chaos** | **in** | **scheme** |  |

APPENDIX A

# About Syntax and Context Rules

This chapter is an extract from the RSL text book. Therefore, some of the mentioned syntactic categories do not exist in the grammar for the RSL subset presented in the present document.

## Syntax Conventions

The syntax contains one or more syntax rules each of the form:

category name ::= alternative1*|*

...

alternativen

where **n** *≥* 1. This rule introduces syntax category (non-terminal) named category name and defines that category as the union of the strings generated by the alternatives. As an example consider:

set type expr ::= finite set type expr*|*

infinite set type expr

Each alternative consists of a sequence of tokens where a token is of one of three kinds:

* A keyword in bold font such as ‘**Bool**’
* A symbol such as ‘(’.
* A subcategory name such as ‘value expr’, possibly prefixed with a text such as ‘*logical-*’ in italics.

The strings generated by an alternative are those obtained by concatenating keywords, symbols and strings from subcategories — in the order of appearance. As examples consider:

finite set type expr ::= type expr**-set**

map type expr ::=

type expr *→m* type expr

The convention below is used for defining optional presence (‘nil-x’ represents absence of ‘x’): For any syntax category name ‘x’ the following rule is assumed:

10

opt-x ::= nil-x*|* x

The conventions below are used for defining repetition: For any syntax category name ‘x’ the following rules are assumed:

x-string ::=

x*|*

x x-string x-list ::=

x*|*

x , x-list x-list2 ::= x , x*|*

x , x-list2

1. choice ::=

x*|*

x *|* x-choice x-choice2 ::=

x *|* x*|*

x *|* x-choice2

Note that ‘*|*’ is an RSL symbol in ‘x *|* x’ and ‘x *|* x-choice2’. x-product2 ::=

x *×* x*|*

x *×* x-product2 x-product ::=

x*|*

x *×* x-product

If ‘opt’ occurs together with ‘string’ or ‘list’, ‘opt’ has the lower precedence. That is, for any syntax category name ‘x’ the following rules are assumed:

opt-x-string ::= nil-x-string*|* x-string

opt-x-list ::= nil-x-list*|* x-list

Similarly, if ‘nil’ occurs together with ‘string’ or ‘list’, ‘nil’ has the lower precedence.

The conventions below are used for indicating context conditions:

If a category name appearing in an alternative is prefixed with a word in italics, then this word indicates a context condition, as explained in the tables A.1–A.3. As an example consider the following syntax rule, where the context condition is that the maximal type of the constituent value expression must be **Bool**:

axiom prefix expr ::=

prefix connective *logical*-value expr

If a category name appearing in an alternative is prefixed with several words in italics separated by underscores, then each of the words indicates a context condition. As an example consider the following syntax rule, where the context conditions are that the constituent value expression must be read-only and have the maximal type **Bool**:

restriction ::=

* *readonly logical*-value expr

If a category name appearing in an alternative is prefixed with a text containing several words in italics separated by ‘ *or* ’, then this text indicates a context condition which is the disjunction of each of the individual context conditions (i.e. one of the context conditions must be fulfilled). As an example consider the following syntax rule, where the context condition is that the constituent name must represent a value or a variable:

value expr ::=

*value or variable*-name

|  |  |
| --- | --- |
| prefix | context condition |
| *unit* | the maximal type of the value expr must be **Unit** |
| *logical* | the maximal type of the value expr must be **Bool** |
| *integer* | the maximal type of the value expr must be **Int** |
| *list* | the maximal type of the value expr must be a list type |
| *map* | the maximal type of the value expr must be a map type |
| *function* | the maximal type of the value expr must be a function type |
| *pure* | the value expr must be pure |
| *readonly* | the value expr must be read-only |

Table A.1: Prefixes of value expr and the context conditions they indicate

|  |  |
| --- | --- |
| prefix | context condition |
| *pure* | the maximal type of the name must be a pure function type |
| *type* | the name must represent a type |
| *value* | the name must represent a value |
| *variable* | the name must represent a variable |
| *channel* | the name must represent a channel |
| *scheme* | the name must represent a scheme |
| *object* | the name must represent an object |

Table A.2: Prefixes of name and the context conditions they indicate

## Static Correctness

This section presents an overview of the common context conditions of RSL.

|  |  |
| --- | --- |
| prefix | context condition |
| *value* | the id must represent a value |

Table A.3: Prefixes of id and the context conditions they indicate

A syntactically correct string is *statically correct* if its context conditions hold. The description *static* indicates that static correctness can be checked decidably (i.e. by a terminating mechanical process). This means that we need to distinguish between what is statically true and what might be proved to be actually true. In particular we need to distinguish between static types (which we call ‘maximal’) and actual types of expressions, and between static accesses (the variables and channels an expression can access) and the actual accesses.

The main context conditions are:

1. All definitions having the same scope must be compatible.
2. All applied occurrences of operators and identifiers must be within the scopes of their definitions and these definitions must be visible.
3. It must be possible to associate uniquely and consistently a ‘maximal’ type with each operator and identifier representing a value, variable or channel. (Note: in RSL\* V1, user defined operators are not allowed.)
4. The accesses to variables and channels of the bodies of explicit function defi- nitions must be allowed by the access descriptions in their signatures. (Note: accesses are not part of RSL\* V1.)
   1. The first condition about compatible definitions prevents things like:

### type

T = **Int**, T = **Bool**

In general different identifiers must be used for different things. There are some exceptions to this — see book chapter on overloading. In RSL\* subset 1, we do not allow overloading.

* 1. The second condition about definitions being visible is similar to the standard rule in block structured programming languages. But note that RSL does not have any ‘define before use’ rule. More details about scope and visibility can be found in RSL book chapter 34.
  2. The third condition about type consistency bans expressions like ‘1 + **true**’.

The rule is concerned with ‘maximal’ types since, for example, it is statically undecidable whether a particular integer expression will evaluate to give a natural number. So it is not against the context conditions to, for example, divide by zero or to take the head of an empty list, but the meaning of such an expression is

typically under-specified in the semantics of RSL (which means that the specifier may not predict how the final implementation will behave).

* 1. The fourth condition ensures that the actual accesses made to variables and channels in the bodies of functions correspond to the accesses allowed in their signatures. So a function is only allowed to read a variable if it has read (or write) access to it, only allowed to write to a variable if it has write access to it, only allowed to input from a channel if it has input access to it and only allowed to output to a channel if it has output access to it.