Doc. ID:
Doc. Ver: 2.1
Doc. Date: 2024-10-23
Page: 1

The SWAN Language Primer (version 2025.0)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Introduction

This document is an excerpt of the Swan language work document [SO-SRS-001] version 1.0 reduced to:

- the syntax;
- illustration of language features that are new compared to Scade 6;
- changes compared to Scade 6.

This reduced version is aimed at practitioners, focusing on illustrating through small examples the new features of the language while the complete document contains more formal descriptions used for Scade One development.

This document beeing extracted from a work document, it is not written to be read linearly; the topics and features in part II are rather independents. It is assumed that the reader is familiar with Scade 6 and its programming paradigm.

Part I
Formalization

1 Syntax of SWAN files

1.1 Notations

Regular expressions notations:

Notation	Signification
ID = *re*	ID is defined by regular expression *re*
|	alternative
re?	0 or 1 occurrence of *re*
*re**	0 occurrence of *re* or more
re+	1 occurrence of X or more
(*re*)	grouping
[*chars*]	matches any of *chars*
[*x*-*y*]	set of chars, ranging from *x* to *y*

Extended Backus-Naur-Form notations:

Notation	Signification
::=	is defined as
|	alternative
⟦ X ⟧	0 or 1 occurrence of X
⦃ X ⦄	0 occurrence of X or more
⦃ X ⦄$^+$	1 occurrence of X or more
⦅ X | Y ⦆	grouping: either X, or Y
abc	keyword terminal symbol abc
INTEGER	terminal symbol other than a keyword
xyz	non-terminal symbol

1.2 Lexical aspects

1.2.1 End of a file

The end of a file is represented by the lexeme $\boxed{\text{EOF}}$.

1.2.2 Comments, markups and spaces

Note about markups: Markups are introduced to provide an escaping mechanism that can be used by a development environment to box snippets of the code for its own needs. The principle is the following: to box a piece of SWAN text, it suffices to select a character sequence "*sequence*" such that neither {*sequence*% nor %*sequence*} appear in the text to box and then add the open markup at the beginning of the snippet and the closed markup at the end. The language interprets these markups to ensure that all the comments starting inside such a box are closed at the end of the box. Markup tokens are defined by:

[S1-132]	*markup*	::=	`{` ALPHANUMERIC* `%`
		|	`%` ALPHANUMERIC* `}`

Considering that markup tokens in a commented zone of text (see requirements below) are ignored, markups must be well-balanced, i.e.:

- an opening markup "`{foo%`" has a corresponding closing one which is the first "`%foo}`" that is afterward in the file;
- if an opening markup "`{foo%`" is encountered then a closing one "`%foo}`" must be present in the file;
- if an open markup "`{foo%`" appears before "`{bar%`", then "`%bar}`" must appear before the matching closing markup "`%foo}`".

Commented zones of text are defined by:

- [S1-001-a] single line comment: starting from `--` and ending at the end of the line or at the closing markup that matches the lastest opened one before the start of the comment, if this closing markup is present in the same line.
- [S1-001-b] multiline comment: all the characters appearing between an open comment separator `/*` and a close comment separator `*/`,
- [S1-001-c] in a multiline comment the closing markup token that matches the last opened one cannot appear in the commented text,
- [S1-001-d] multiline comments can be nested.

[S1-002]
The following symbols are ignored, except within a ASCII literal:

space (' '), *tabulation* ('\t'), *carriage return* ('\r'), *line feed* ('\n') and *form feed* ('\f').

Note: Character codes '\t', '\r', '\n' and '\f' are given using the C language conventions.

1.2.3 Lexemes and Symbols

[S1-003]	DIGIT2	=	[`0-1`]
[S1-004]	DIGIT8	=	[`0-7`]
[S1-005]	DIGIT10	=	[`0-9`]
[S1-006]	DIGIT16	=	[`a-f`] | [`A-F`] | [`0-9`]

```
[S1-007]          INTEGER2 = 0b DIGIT2+

[S1-008]          INTEGER8 = 0o DIGIT8+

[S1-009]         INTEGER10 = 0 | ( [ 1-9 ] DIGIT10* )

[S1-010]         INTEGER16 = 0x DIGIT16+

[S1-011]           INTEGER = INTEGER2
                           | INTEGER8
                           | INTEGER10
                           | INTEGER16

[S1-012]     TYPED__INTEGER = INTEGER (_i | _ui) (8 | 16 | 32 | 64)

[S1-013]         EXPONENT = [ eE] [+-]? DIGIT10+
                    FLOAT = DIGIT10+ . DIGIT10* EXPONENT?
                          | DIGIT10* . DIGIT10+ EXPONENT?

[S1-014]      TYPED__FLOAT = FLOAT (_f32 | _f64)

[S1-015]           LETTER = [ a-z ] | [ A-Z ]
             ALPHANUMERIC = DIGIT10 | LETTER | _
                     WORD = LETTER ALPHANUMERIC*
                    ASCII = ALPHANUMERIC | SPACE
                          | [!"$#%&'()*+,-./:; <=> ?@[] \ ^`|{}~]
                     HEXA = \x DIGIT16 DIGIT16
                CHARACTER = ASCII | HEXA

[S1-016]             CHAR = 'CHARACTER'

[S1-017]               ID = WORD

[S1-018]             NAME = ' ID

[S1-020]             LUID = $ ID

[S1-211]            LUNUM = # DIGIT10+
```

[S1-024]
The following symbols are recognized as lexemes:

```
<>   <=   >=   <<   >>   =   <   >
(    )    [    ]    {    }
::   ..   :>   ;    ,    .   :   __   =>   |
->   ^    @
+    -    *    /    \
```

`[S1-130]`

The following pairs (char, hex) of CHAR are equivalent:

char	hex	char	hex	char	hex	char	hex	char	hex
`' '`	`'\x20'`	`'3'`	`'\x33'`	`'F'`	`'\x46'`	`'Y'`	`'\x59'`	`'l'`	`'\x6C'`
`'!'`	`'\x21'`	`'4'`	`'\x34'`	`'G'`	`'\x47'`	`'Z'`	`'\x5A'`	`'m'`	`'\x6D'`
`'"'`	`'\x22'`	`'5'`	`'\x35'`	`'H'`	`'\x48'`	`'['`	`'\x5B'`	`'n'`	`'\x6E'`
`'#'`	`'\x23'`	`'6'`	`'\x36'`	`'I'`	`'\x49'`	`'\'`	`'\x5C'`	`'o'`	`'\x6F'`
`'$'`	`'\x24'`	`'7'`	`'\x37'`	`'J'`	`'\x4A'`	`']'`	`'\x5D'`	`'p'`	`'\x70'`
`'%'`	`'\x25'`	`'8'`	`'\x38'`	`'K'`	`'\x4B'`	`'^'`	`'\x5E'`	`'q'`	`'\x71'`
`'&'`	`'\x26'`	`'9'`	`'\x39'`	`'L'`	`'\x4C'`	`'_'`	`'\x5F'`	`'r'`	`'\x72'`
`'''`	`'\x27'`	`':'`	`'\x3A'`	`'M'`	`'\x4D'`	``'`'``	`'\x60'`	`'s'`	`'\x73'`
`'('`	`'\x28'`	`';'`	`'\x3B'`	`'N'`	`'\x4E'`	`'a'`	`'\x61'`	`'t'`	`'\x74'`
`')'`	`'\x29'`	`'<'`	`'\x3C'`	`'O'`	`'\x4F'`	`'b'`	`'\x62'`	`'u'`	`'\x75'`
`'*'`	`'\x2A'`	`'='`	`'\x3D'`	`'P'`	`'\x50'`	`'c'`	`'\x63'`	`'v'`	`'\x76'`
`'+'`	`'\x2B'`	`'>'`	`'\x3E'`	`'Q'`	`'\x51'`	`'d'`	`'\x64'`	`'w'`	`'\x77'`
`','`	`'\x2C'`	`'?'`	`'\x3F'`	`'R'`	`'\x52'`	`'e'`	`'\x65'`	`'x'`	`'\x78'`
`'-'`	`'\x2D'`	`'@'`	`'\x40'`	`'S'`	`'\x53'`	`'f'`	`'\x66'`	`'y'`	`'\x79'`
`'.'`	`'\x2E'`	`'A'`	`'\x41'`	`'T'`	`'\x54'`	`'g'`	`'\x67'`	`'z'`	`'\x7A'`
`'/'`	`'\x2F'`	`'B'`	`'\x42'`	`'U'`	`'\x55'`	`'h'`	`'\x68'`	`'{'`	`'\x7B'`
`'0'`	`'\x30'`	`'C'`	`'\x43'`	`'V'`	`'\x56'`	`'i'`	`'\x69'`	`'|'`	`'\x7C'`
`'1'`	`'\x31'`	`'D'`	`'\x44'`	`'W'`	`'\x57'`	`'j'`	`'\x6A'`	`'}'`	`'\x7D'`
`'2'`	`'\x32'`	`'E'`	`'\x45'`	`'X'`	`'\x58'`	`'k'`	`'\x6B'`	`'~'`	`'\x7E'`

Note: literals `'\x`d_1d_2`'` and `'\x`d_3d_4`'` are equivalent if d_1d_2 and d_3d_4 differ only by their case.

1.2.4 Keywords

`[S1-025]`

abstract, **activate**, **and**, **as**, **assume**, **assert**, **automaton**,
bool, **block**, **byname**, **bypos**,
case, **char**, **clock**, **const**,
def, **default**, **diagram**, **do**,
else, **elsif**, **emit**, **end**, **enum**, **every**, **expr**
false, **fby**, **final**, **flatten**, **float**, **float32**, **float64**, **fold**, **foldi**, **foldw**, **foldwi**, **forward**,
forwardi, **function**,
guarantee, **group**,
if, **imported**, **initial**, **inline**, **int8**, **int16**, **int32**, **int64**, **integer**, **is**,
land, **lnot**, **lor**, **lsl**, **lsr**, **lxor**, **last**, **let**,
make, **map**, **mapfold**, **mapfoldi**, **mapfoldw**, **mapfoldwi**, **mapi**, **mapw**, **mapwi**,
match, **merge**, **mod**,
node, **not**, **numeric**,
of, **onreset**, **open**, **or**,
pack, **package**, **parameter**, **pre**, **private**, **probe**, **public**,
repeat, **repeati**, **restart**, **resume**, **returns**, **reverse**,
self, **sensor**, **sig**, **signed**, **specialize**, **state**, **synchro**,
tel, **then**, **times**, **transpose**, **true**, **type**,
uint8, **uint16**, **uint32**, **uint64**, **unless**, **unsigned**, **until**, **use**,
var,
when, **where**, **window**, **wire**, **with**,
xor

Notice that the following keywords are reserved but unused: **abstract**, **do**, **fby**, **final**, **foldw**, **foldwi**, **forwardi**, **imported**, **is**, **make**, **mapfoldw**, **mapfoldwi**, **mapw**, **mapwi**, **onreset**, **open**, **package**, **parameter**, **private**, **public**, **repeat**, **repeati**, **sig**, **synchro**, **tel**, **times**.

1.2.5 Pragmas

Pragmas are a mean to pass information to tools without having to follow the language semantics. The pragma concept belongs to the language, but the pragmas themselves do not.

Pragmas syntax is:

`[S1-203]` *pragma* ::= **#pragma** CHARACTER* **#end**

Pragmas contain any kind of character and can be multilines.

- Inside such pragmas, a doubled '`#`' character ('`##`') is interpreted as a single '`#`' character. This allows to write "`##end`", which will not close the pragma and will result in the sequence "`#end`" in the pragma. Note that a single '`#`' character is interpreted as itself.
- The closing markup token (see 1.2.2) that matches the last one opened before **#pragma** cannot appear in the text of this pragma.

The syntax accepted and/or recognized for the pragmas shall be detailed by the tools that define them.

1.3 Declarations

[S1-027] *module_body_file* ::= ⦃ *use_clause* ⦄
⦃ *body_decls* ⦄
$\boxed{\text{EOF}}$

[S1-185] *module_interface_file* ::= ⦃ *use_clause* ⦄
⦃ *interface_decls* ⦄
$\boxed{\text{EOF}}$

[S1-186] *use_clause* ::= **use** *path_id* ⟦ **as** ID ⟧ **;**

[S1-029] *path_id* ::= ⦃ ID **::** ⦄ ID

[S1-028] *body_decls* ::= *types_and_globals*
| *user_op_decl*

[S1-187] *interface_decls* ::= *types_and_globals*
| *user_op_interface*

[S1-188] *types_and_globals* ::= **group** ⦃ *group_decl* **;** ⦄
| **type** ⦃ *type_decl* **;** ⦄
| **const** ⦃ *const_decl* **;** ⦄
| **sensor** ⦃ *sensor_decl* **;** ⦄

Types

[S1-033] *type_decl* ::= ID ⟦ **=** *type_def* ⟧

[S1-136] *type_def* ::= *type_expr*
| **enum** **{** ID ⦃ **,** ID ⦄ **}**
| *variant* ⦃ **|** *variant* ⦄
| *struct_texpr*

[S1-034] *type_expr* ::= **bool**
| **signed** << *expr* >>
| **int8** | **int16** | **int32** | **int64**
| **unsigned** << *expr* >>
| **uint8** | **uint16** | **uint32** | **uint64**
| **float32** | **float64**
| **char**
| *path_id*
| *typevar*
| *type_expr* ^ *expr*

[S1-035] *struct_texpr* ::= **{** *field_decl* ⦃ **,** *field_decl* ⦄ **}**

[S1-036] *field_decl* ::= ID **:** *type_expr*

[S1-037] *variant* ::= ID *variant_type_expr*

[S1-038] *variant_type_expr* ::= **{** ⟦ *type_expr* ⟧ **}**
| *struct_texpr*

[S1-039] *typevar* ::= NAME

Groups

[S1-041] *group_decl* ::= ID **=** *group_type_expr*

[S1-042] *group_type_expr* ::= *type_expr*
| **(** *group_type_expr* ⦃ **,** *group_type_expr* ⦄
⦃ **,** ID **:** *group_type_expr* ⦄ **)**
| **(** ID **:** *group_type_expr* ⦃ **,** ID **:** *group_type_expr* ⦄ **)**

Globals

[S1-043] *const_decl* ::= ID **:** *type_expr* ⟦ **=** *expr* ⟧
| ID **=** *expr*

[S1-045] *sensor_decl* ::= ID **:** *type_expr*

1.4 User defined operators

Variable declarations

[S1-046] *var_decl* ::= *var_id* ⟦ **:** *group_type_expr* ⟧ ⟦ **when** *clock_expr* ⟧
⟦ **default** **=** *expr* ⟧
⟦ **last** **=** *expr* ⟧

[S1-047] *var_id* ::= ⟦ **clock** ⟧ ID

Operators

[S1-048]	*user_op_decl*	::=	⟦ **inline** ⟧ *op_kind* ID ⟦ *size_decl* ⟧ *params* **returns** *params* ⦃ *where_decl* ⦄ ⟦ *spec_decl* ⟧ *opt_body*
[S1-189]	*user_op_interface*	::=	⟦ **inline** ⟧ *op_kind* ID ⟦ *size_decl* ⟧ *params* **returns** *params* ⦃ *where_decl* ⦄ ⟦ *spec_decl* ⟧ **;**
[S1-049]	*op_kind*	::=	**function** | **node**
[S1-040]	*size_decl*	::=	<< ID ⦃ **,** ID ⦄ >>
[S1-051]	*params*	::=	(⟦ ⦃ *var_decl* **;** ⦄ *var_decl* ⟦ **;** ⟧ ⟧)
[S1-052]	*where_decl*	::=	**where** *typevar* ⦃ **,** *typevar* ⦄ *numeric_kind*
[S1-137]	*numeric_kind*	::=	**numeric** | **float** | **integer** | **signed** | **unsigned**
[S1-053]	*spec_decl*	::=	**specialize** *path_id*
[S1-054]	*opt_body*	::=	**;**
		|	*data_def*

Scopes

[S1-056]	*data_def*	::=	*equation* **;**
		|	*scope*
[S1-058]	*scope*	::=	**{** *scope_sections* **}**
[S1-134]	*scope_sections*	::=	⦃ *scope_section* ⦄
[S1-059]	*scope_section*	::=	**var** ⦃ *var_decl* **;** ⦄
		|	**let** ⦃ *equation* **;** ⦄
		|	**emit** ⦃ ⟦ LUID ⟧ *emission_body* **;** ⦄
		|	*assert_kind* ⦃ LUID **:** *expr* **;** ⦄
		|	**diagram** ⦃ *object* ⦄
[S1-210]	*assert_kind*	::=	**assume** | **assert** | **guarantee**

Equations

[S1-060]	*equation*	::=	*lhs* ⟦ LUID ⟧ = *expr*
		|	*def_by_case*
[S1-062]	*lhs*	::=	**( )**
		|	*lhs_item* ⦃ **,** *lhs_item* ⦄ ⟦ **, ..** ⟧
[S1-063]	*lhs_item*	::=	ID
		|	**_**
[S1-064]	*def_by_case*	::=	⟦ *lhs* **:** ⟧ ⦅ *state_machine* | *select_activation* ⦆

Emissions

[S1-065]	*emission_body*	::=	*flow_names* ⟦ **if** *expr* ⟧
[S1-067]	*flow_names*	::=	NAME ⦃ **,** NAME ⦄

Activations

[S1-068]	*select_activation*	::=	**activate** ⟦ LUID ⟧
			⦅ *if_activation* | *match_activation* ⦆
[S1-069]	*match_activation*	::=	**when** *expr* **match**
			⦃ **|** *pattern_with_capture* **:** *data_def* ⦄$^+$
[S1-070]	*if_activation*	::=	**if** *expr* **then** *ifte_branch*
			⦃ **elsif** *expr* **then** *ifte_branch* ⦄
			else *ifte_branch*
[S1-071]	*ifte_branch*	::=	*data_def*
		|	*if_activation*

State machines

[S1-073]	*state_machine*	::=	**automaton** ⟦ LUID ⟧
			⦃ *state_decl* | *transition_decl* ⦄
[S1-074]	*state_decl*	::=	⟦ **initial** ⟧ **state** *state_id* **:**
			⟦ **unless** ⦃ *transition* ⦄$^+$ ⟧
			scope_sections
			⟦ **until** ⦃ *transition* ⦄$^+$ ⟧
[S1-072]	*state_id*	::=	ID
		|	LUNUM ⟦ ID ⟧
[S1-075]	*transition_decl*	::=	*priority* *state_ref* ⦅ **unless** | **until** ⦆ *transition*
[S1-076]	*priority*	::=	**:** ⟦ INTEGER ⟧ **:**
[S1-213]	*state_ref*	::=	LUNUM | ID

[S1-077]	*transition*	::=	**if** *guarded_arrow* **;**
		|	⟦ *scope* ⟧ *target* **;**
[S1-078]	*guarded_arrow*	::=	**(** *expr* **)** *arrow*
[S1-079]	*arrow*	::=	⟦ *scope* ⟧ ⦅ *target* | *fork* ⦆
[S1-080]	*target*	::=	⦅ **restart** | **resume** ⦆ *state_ref*
[S1-081]	*fork*	::=	**if** *guarded_arrow*
			⦃ **elsif** *guarded_arrow* ⦄
			⟦ **else** *arrow* ⟧
			end
		|	⦃ *fork_priority* ⦄ **end**
[S1-082]	*fork_priority*	::=	*priority* **if** *guarded_arrow*
		|	*priority* **else** *arrow*

Diagrams

[S1-084]	*object*	::=	**(** *description* ⟦ *local_objects* ⟧ **)**
[S1-178]	*local_objects*	::=	**where** ⦃ *object* ⦄

[S1-085]	*description*	::=	⟦ LUNUM ⟧ ⟦ LUID ⟧ *graph_item*
		|	*def_by_case*
		|	*scope_section*
[S1-212]	*graph_item*	::=	**expr** *expr*
		|	**def** *lhs*
		|	**block** *operator_block*
		|	**group** ⟦ *group_operation* ⟧
		|	**wire** *connection* **=>** *connection* ⦃ **,** *connection* ⦄

[S1-179]	*group_operation*	::=	**()**
		|	**byname**
		|	**bypos**

[S1-087]	*connection*	::=	*port* ⟦ *group_adaptation* ⟧
		|	**()**
[S1-088]	*port*	::=	*instance_id*
[S1-090]	*instance_id*	::=	LUNUM | LUID | **self**

1.5 Expressions

[S1-091] *expr* ::= *id_expr*
| *atom*
| *unary_op* *expr*
| *expr* *binary_op* *expr*
| *expr* **when** *clock_expr*
| *expr* **when match** *path_id*
| **(** *expr* **:>** *type_expr* **)**
| *group_expr*
| *composite_expr*
| *switch_expr*
| *fwd_expr*
| *operator_instance* **(** *group* **)**
| *port*
| *multigroup_prefix*

[S1-092] *id_expr* ::= *path_id*
| **last** NAME

[S1-094] *atom* ::= **true** | **false**
| CHAR
| INTEGER | TYPED_INTEGER
| FLOAT | TYPED_FLOAT

[S1-095] *unary_op* ::= **−** | **+** | **lnot**
| **not**
| **pre**

[S1-096] *binary_op* ::= **+** | **−** | ***** | **/** | **mod** | **land** | **lor** | **lxor** | **lsl** | **lsr**
| **=** | **<>** | **<** | **>** | **<=** | **>=**
| **and** | **or** | **xor**
| **->** | **pre**
| **@**

[S1-181] *n_ary_op* ::= **+** | ***** | **@** | **and** | **or** | **xor** | **land** | **lor**

[S1-182] *multigroup_prefix* ::= **window** *size* **(** *group* **)** **(** *group* **)**
| **merge** **(** *group* **)** {{ **(** *group* **)** }}

[S1-184] *size* ::= **<<** *expr* **>>**

[S1-097] *clock_expr* ::= ID
  | **not** ID
  | (ID **match** *pattern*)

[S1-098] *group_expr* ::= (*group*)
  | *expr* *group_adaptation*
[S1-099] *group* ::= ⟦ *group_item* ⦃ , *group_item* ⦄ ⟧
[S1-100] *group_item* ::= ⟦ ID **:** ⟧ *expr*

[S1-101] *group_adaptation* ::= **.** (*group_renamings*)
[S1-135] *group_renamings* ::= ⟦ *renaming* ⦃ , *renaming* ⦄ ⟧
[S1-102] *renaming* ::= ((ID | INTEGER)) ⟦ **:** ⟦ ID ⟧ ⟧

[S1-104] *composite_expr* ::= *expr* *label_or_index*
  | *path_id* **group** (*expr*)
  | *expr* [*expr* **..** *expr*]
  | (*expr* **.** ⦃ *label_or_index* ⦄$^+$ **default** *expr*)
  | *expr* ^ *expr*
  | [*group*]
  | *struct_expr*
  | *variant_expr*
  | (*expr* **with** *modifier* ⦃ ; *modifier* ⦄ ⟦ ; ⟧)
[S1-105] *struct_expr* ::= **{** *group* **}** **:** *path_id*
[S1-093] *variant_expr* ::= *path_id* **{** *group* **}**
[S1-106] *modifier* ::= ⦃ *label_or_index* ⦄$^+$ = *expr*
[S1-108] *label_or_index* ::= **.** ID
  | [*expr*]

[S1-110] *switch_expr* ::= **if** *expr* **then** *expr* **else** *expr*
  | (**case** *expr* **of** ⦃ **|** *pattern* **:** *expr* ⦄$^+$)

```
[S1-111]                  pattern ::= path_id
                                    | path_id _
                                    | path_id { }
                                    | CHAR
                                    | ⟦ − ⟧ INTEGER
                                    | ⟦ − ⟧ TYPED_INTEGER
                                    | true | false
                                    | _
                                    | default
[S1-112]     pattern_with_capture ::= pattern
                                    | path_id { ID }

[S1-191]     fwd_expr ::= forward ⟦ LUID ⟧⟦ ⦅ restart | resume ⦆ ⟧ ⦃ dim ⦄+
                          fwd_body
                          returns ( returns_group )

[S1-192]     fwd_body ::= ⟦ unless expr ⟧
                            scope_sections
                          ⟦ until expr ⟧

[S1-193]              dim ::= size
                              ⟦ with ⦅ << ID >> | current_elt ⦆ ⦃ current_elt ⦄ ⟧
[S1-195]      current_elt ::= current_lhs = expr ;
[S1-196]      current_lhs ::= ID
                            | [ current_lhs ]

[S1-194]    returns_group ::= ⟦ returns_item ⦃ , returns_item ⦄ ⟧
[S1-197]     returns_item ::= item_clause
                            | ⟦ ID = ⟧ array_clause
[S1-198]   returns_clause ::= ⦅ item_clause | array_clause ⦆
[S1-199]      item_clause ::= ID ⟦ : last_default ⟧
[S1-200]     last_default ::= last = expr
                            | default = expr
                            | last = expr default = expr
                            | last = default = expr
[S1-201]     array_clause ::= [ returns_clause ]
```

Operator instance expression

[S1-114] *operator_block* ::= *operator*
| *op_expr*

[S1-115] *operator_instance* ::= *operator* ⟦ LUID ⟧
[S1-116] *operator* ::= *prefix_op* ⟦ *sizes* ⟧
[S1-183] *sizes* ::= << *expr* ⦃ , *expr* ⦄ >>

[S1-117] *prefix_op* ::= *path_id*
| *prefix_primitive*
| **(** *op_expr* **)**

[S1-118] *op_expr* ::= *iterator* *operator*
| **activate** *operator* **every** *clock_expr*
| **activate** *operator* **every** *expr* ⦅ **last** | **default** ⦆ *expr*
| **restart** *operator* **every** *expr*
| *op_kind* *anonymous_op*
| *operator* \ *partial_group*
| *n_ary_op*

[S1-119] *anonymous_op* ::= *params* **returns** *params* *data_def*
| ID ⦃ , ID ⦄ *scope_sections* **=>** *expr*

[S1-120] *prefix_primitive* ::= **reverse**
| **transpose** ⟦ **{**⟦ INTEGER **,** ⟧ INTEGER **}** ⟧
| **pack**
| **flatten**

[S1-121] *iterator* ::= **map** | **fold** | **mapfold**
| **mapi** | **foldi** | **mapfoldi**

[S1-125] *partial_group* ::= *opt_group_item* ⦃ , *opt_group_item* ⦄
[S1-126] *opt_group_item* ::= _ | *group_item*

Primitive operators associativity and relative priority

[S1-129]
Expression operators from the higher priority to the lowest:

. [ ]	
@	
^	
not lnot	
when	
+ − (unary)	
pre	right associative
*** / mod**	
+ − (binary)	
lsl lsr	
= <> <= >= < >	
and	
or xor	
land	
lor lxor	
->	right associative
if	

Except for **->** and **pre** all the infix operators are left associative.

2 Post-syntactic requirements

2.1 Post-syntactic rules

The following rules correspond to syntactic forms that are mandatory but not covered by the Backus-Naur-Form, in order to alleviate it.

Operators

- `[S1-055-a]` Input and output declarations in an operator interface shall specify a type. This restriction holds for all the operators specifying a complete interface (inputs and outputs) whether they are named or anonymous, thus the short form of anonymous operators is not concerned.
- `[S1-055-b]` The output (resp. input) declarations of an operator declaration in a module interface (*user_op_interface*) shall not specify last or default (resp. last) values.
- `[S1-055-c]` The output (resp. input) declarations of an operator declaration in a module body (*user_op_decl*) in the case the operator has no body shall not specify last or default (resp. last) values.

Operator expression

- `[S1-206-a]` In an operator expression "**activate** *op* **every** ..." the sub-expression *op* shall not contain another **activate**;
- `[S1-206-b]` In an operator expression "**restart** *op* **every** ..." the sub-expression *op* shall contain neither **activate** nor **restart**.
- `[S1-206-c]` In an operator expression "*iterator op*" the sub-expression *op* shall contain neither **activate** nor **restart**.

Note: the meaning of "the sub-expression *op* shall not contain *X*" is to be understood as *X* shall not appear when recursively unfolding the *operator* part of the *op_expr* case that applies.

Arrays

- `[S1-140-a]` An array cannot be defined by an empty list.

Definition by case

- `[S1-202-a]` The left-hand side of a definition by case is of the form `()` or $id, \ldots, id$.

State Machines

- `[S1-204-a]` If an automaton has declared a transition (*transition_decl*) with a given source state, then the declaration of this state (*state_decl*) cannot contain transitions.
- `[S1-204-b]` The transitions in an automaton shall all be of the same kind i.e. either *weak* (**until**) or *strong* (**unless**). [1]

[1] Note that this restriction does not affect the nesting of automata, a state can have strong or weak transitions and contain automata with strong or weak transitions.

- `[S1-204-c]` When a state appears as the source of more than one declared outgoing transition (*transition_decl*), these outgoing transitions shall all have an explicit priority.
- `[S1-204-d]` Declared outgoing transitions (*transition_decl*) for a given source state shall have pairwise different priorities.
- `[S1-204-e]` All the elements of a declared fork list (list of *fork_priority*) in a *fork* transition shall have an explicit priority.
- `[S1-204-f]` A declared fork list (list of *fork_priority*) in a *fork* transition shall have all its elements with pairwise different priorities.
- `[S1-204-g]` A declared fork list (list of *fork_priority*) in a *fork* transition shall have at most one **else** branch.
- `[S1-204-h]` An **else** branch in a declared fork list (list of *fork_priority*) in a *fork* transition shall have the lowest priority.
- `[S1-204-i]` A declared fork list (list of *fork_priority*) in a fork branch (rule *fork*) shall contain at least one element.
- `[S1-204-j]` A declared fork list (list of *fork_priority*) in a *fork* that contains one **else** branch shall contain at least one other branch.

Forward iterations

- `[S1-207-a]` A partial forward, i.e. a forward with an **unless** or an **until** or both, shall not specify more than one dimension.

2.2 Unicity of local identifiation

In the syntax specification the tokens LUID (locally unique identifier) and LUNUM (locally unique number) appear in several places. The considered *locality* is the one of the operator body the LUID or LUNUM appears in. These token shall allow a direct (no possible masking effect), non ambiguous and identification of the construct it designates in the operator definition.

The rules that may introduce a new LUID are :

- *select_activation*
- *state_machine*
- *description*
- *fwd_expr*
- *operator_instance*
- *equation*
- *scope_section*

The rules that may introduce a new LUNUM :

- *identification*
- *description*
  - [S1-205-a] All the LUID present in a given operator definition (*user_op_def* with a body present) shall be unique within this definition.
  - [S1-205-b] All the LUNUM present in a given operator definition (*user_op_def* with a body present) shall be unique within this definition.

Part II
Extensions

3 Groups

Groups in Swan are a strict extension of Scade 6 groups allowing to have both positional and named items. For instance here is a pure positional group expression `(1, not b, x*y)` and here a completely named one `(rank: 1, validity: not b, size: x*y)`. In the first only the position matters, permuting the fields would lead to a change in the way flows are related in the program and most of the time in a type error, which is not the case in the second since every sub-expression is associated to a name, fields can be permuted without affecting the ability to refer to these components. So named components are identified by the name of the field only, their position does not matter. It is sometimes convenient to mix positional and named fields, typically in operator application which definition proposes default values for some of its inputs, the first parameters can be specified using a positional convention until the first omission (filled with the default), then the labels are used for the other parameters. As in Scade 6, it is allowed to build a group with groups by juxtaposing them between parentheses and separated by commas. The interpretation of this new group is based on the one of the original groups and on the kind of the fields:

- positional ones are flattened and their new position is given by the relative position of their original group and their position inside this original group;
- named ones are just unioned provided that there is no field name collision (collisions are statically checked and models with collisions rejected).

For instance, the group `((e, f, l1:(l2:a2, l3:b2, d1)), (a, l4:c1, b))` where the `l1`,...,`l4` are field labels and the other are Swan expressions is interpreted as `(e, f, a, b, l1:(d1, l2:a2, l3:b2), l4:c1)` which is itself equivalent to `(e, f, a, b, l4:c1, l1:(d1, l2:a2, l3:b2))` because named fields are not ordered. So the interpretation of a group expression is a pair formed by the list of ordered fields and the set of named fields. Since there is no name collision in the fields name, given a total order on the labels, we can define a normal form for the group structure that applies either to group expressions or group types.

4 Modules and namespaces

Swan proposes *namespaces* and *modules* to organize model and libraries. The structure is the following:

- a namespace can contain other namespaces and modules;
- a module contains groups, types, constants, sensors and operators;
- there is an implicit top namespace that is not named.

Figure 1 illustrates the use of namespaces (in blue) to organize the different modules that define the libraries used by the application. The application is itself in a module but not in a specific namespace. This organization allows to use libraries providing modules with the same name (here Matrix) as long as they are not in the same namespace.

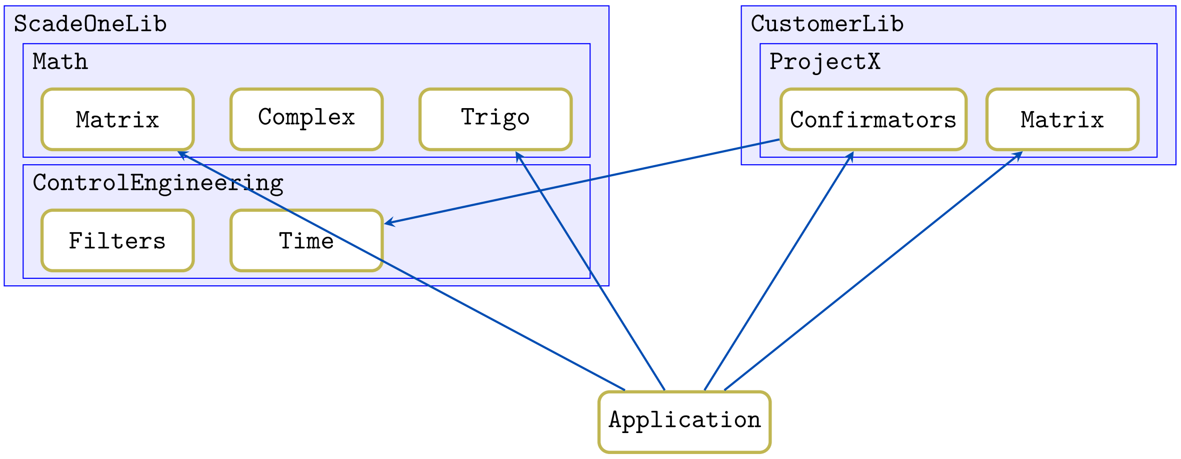


Figure 1: Example of a project organization using namespaces

4.1 Modules

4.1.1 Definition

A module is defined by at least one of these two elements:

1. a module interface file whose syntax is defined by rule *module_interface_file* and
2. a collection of module body files, each file follows the syntax defined by rule *module_body_file*.

These files have the specific extensions:

- `file.swani` for an interface and
- `file.swan` for a body.

Terminology:

- *declaration*: refers to the introduction of an element with its kind (type, constant, ...) and the typing information to check its correct usage; a declaration informs about the existence of an element and not about its implementation.
- *definition*: refers to the introduction of an element with the information provided by a declaration extended with all the necessary implementation details.

Declarations contained in the interface can be referred to by any other module, provided that a dependency has been explicitly specified using a `use` directive as explained in 4.3. In case no interface file is provided, all declarations contained in the body files can be accessed.

The declarations and definitions present in a module interface can only reference the elements of the module present in the interface itself and those that are visible in the modules referenced in the `use` directives of the interface. The body of the module is in the scope of the `use` directive specified in the module interface, if any. All the elements in the module interface, if any, are usable in the body of the module. All the elements introduced in a body file are accessible in all the body files contributing to the same module.

4.1.2 Consistency

Declared, defined and imported:

- a module element can be declared only once, either in the interface or in the body;
- a module element can be defined only once, either in the interface or in the body;
- a module element declared in the interface can be defined in the body;
- a module element that is declared and not defined is imported.

As a consequence of the previous points, a module defined by an interface file only, has all its undefined elements considered imported.

Naming: A module introduces a fresh naming environement for all the kind of elements it declares or defines, i.e.: groups, types, constants, operators.

Declaration restrictions

- an operator declaration shall not specify last values for its I/Os.
- an operator declaration shall not specify default values for its outputs.

Declaration/definition consistency: In presence of both a declaration and a definition for a given module element both information shall be consistent which is guaranteed by the following checks :

- constants have a declaration type compatible with the definition type.
- operators have the same kind (**node**/**function**) and the same declared inlining (**inline**) in both definition and declaration. The declared I/O for the operator also satisfy:
  - same input groups with same input default values;
  - same output groups (defined output last and default values are ignored);
  - corresponding types in the leafs of the I/O groups are equivalent.
  - same number of size parameters with same names.
  - same number of type variables with same name and same constraints given in the same order.

Note that the type for an operator is a function type from the operator input group to its output group and that these I/O groups embed the identifiers used for the formal parameters; this means that, to be compatible, the declaration and definition must agree on the formal parameter names.

Here is an example, let us consider a module `M` has the following interface:

```
node Confirmator <<N>> (a: 'T default = 0) returns (o: bool)
  where 'T numeric;
```

The following signatures of implementations are **consistent** with this interface:

```
node Confirmator <<N>> (a: 'T default = 0) returns (o: bool)
  where 'T numeric {
...
}
```

```
node Confirmator <<N>> (a: 'T default = 0 last = 12) returns (o: bool )
  where 'T numeric {
...
}
```

```
node Confirmator <<N>> (a: 'T default = 0) returns (o: bool default = false)
  where 'T numeric {
...
}
```

The following signatures of implementations are **not consistent** with this interface:

```
node Confirmator <<N>> (a: 'T default = 42) returns (o: bool) where 'T numeric {
...
}
```

```
node Confirmator <<N>> (a: 'T) returns (o: bool) where 'T numeric {
...
}
```

```
node Confirmator <<P>> (a: 'T default = 0) returns (o: bool) where 'T numeric {
...
}
```

```
node Confirmator <<N>> (a: 'U) returns (o: bool) where 'U numeric {
...
}
```

```
node Confirmator <<N>> (a: 'T default = 0) returns (o: bool ) where 'T unsigned {
...
}
```

(**TODO: all** *to be defined for all the type systems*). ⇐

Restriction A module `M` cannot refer to its own content through a qualified path.
For instance:
`M.swani`

```
...

type T;

function f (x: M::T) returns (y: M::T);
...
```

is incorrect and shall be written:

`M.swani`

```
...

type T;

function f (x: T) returns (y: T);
...
```

The reason for introducing this restriction is because qualified names are giving access to a module content through its interface which is confusing in particular when used in a module body where all these elements are visible.

4.2 Organizing namespaces

Namespaces are introduced by the structure of the filenames that define the modules (`.swan` and `.swani`).

BASENAME	=	⦃ NAMESPACE`-` ⦄MODULE
INTERFACE__FILENAME	=	BASENAME`.swani`
BODY__FILENAME	=	BASENAME⟦ `.`CONTRIB ⟧`.swan`
NAMESPACE	=	WORD
MODULE	=	WORD
CONTRIB	=	ALPHANUMERIC*

The interface and the body contributions of a module agree on the BASENAME. Based on this convention, the hierarchical organization of figure 1 can be captured with the following list of files:

```
Application.swan
CustomerLib-ProjectX-Confirmators.swani
CustomerLib-ProjectX-Matrix.part1.swan
CustomerLib-ProjectX-Matrix.part2.swan
ScadeOneLib-Math-Matrix.swani
ScadeOneLib-Math-Matrix.swan
ScadeOneLib-Math-Complex.swani
ScadeOneLib-Math-Complex.swan
ScadeOneLib-Math-Trigo.swani
ScadeOneLib-Math-Trigo.swan
ScadeOneLib-ControlEngineering-Filters.swani
ScadeOneLib-ControlEngineering-Filters.1.swan
ScadeOneLib-ControlEngineering-Filters.2.swan
ScadeOneLib-ControlEngineering-Time.swani
ScadeOneLib-ControlEngineering-Time.swan
```

Note: it is allowed to have a module and a namespace at same level in a namespace/module hierarchy since it is never ambiguous in the context.

4.3 Using a module from a namespaces

To use a module that is in a namespace (either named or top level) in an interface or body of another module, it must first be introduced by a "`use` *path_id*" directive where the *path_id* is an absolute path from the top level namespace to the name of the module to be used. For instance in figure 1, the module `Application` that contains the root operator of the application shall specify in its body that it needs `Trigo` module:

Application.swan
```
...
use ScadeOneLib::Math::Trigo;

group point = (x: float64, y: float64);

function to_cartesian (r: float64; teta: float64) returns (p: point)
  p = (r * Trigo::cos(teta), r * Trigo::sin(teta));

...
```

We can see that the `use` directive gives the path to a module from the top namespace and makes it available in the rest of the module that specifies the `use`. Namespaces only appear in the filenames as specified in 4.2 and in the `use` directive. The `use` directive can also be complemented with a renaming to locally introduce a shorter name, for instance:

Application.swan
```
...
use ScadeOneLib::Math::Trigo as Trg;

group point = (x: float64, y: float64);

function to_cartesian (r: float64; teta: float64) returns (p: point)
  p = (r * Trg::cos(teta), r * Trg::sin(teta));

...
```

It can also serve to solve a local conflict introduced by the use of two modules with the same name and from different namespaces:

Application.swan
```
use ScadeOneLib::Math::Matrix as SMatrix;
use CustomerLib::ProjectX::Matrix as PXMatrix;
...
```

If an interface (.swani) specification uses an element from another module, it must contains the appropriate use. A module interface file is in the scope of the `use`s it specifies, but not in those specified in the associated body files. A module body file is in the scope of all the `use`s specified either in the interface or in another body contribution file of the same module. In a module body or interface a used module can be aliased with different names; as long as they resolve to the same module in the same namespace, they can be used interchangeably.

4.4 Notes about modules and separate compilation

4.4.1 Module dependencies

A module can be used by another module based on the interface only (i.e. without the implementation details given in the body) and the different static checks can be done with respect to what it is declared in the interfaces.

However it is not possible to achieve the non circularity of the definition without involving all the definitions since the interface does not give this kind of detail which is expected for the level of abstraction it offers. Thus in the case of a separate compilation process, the circularity of the definitions may only appear late, when assembling all the compilation products to build the application. This means that a separate compilation of SWANis possible but requires that each compile module come with an information, produced by the compiler, about the dependencies to other modules; then a link phase must verify the absence of circularity.

4.4.2 Inline operators

A module that contains operators specified as **inline** cannot be compiled separately and must be distributed with the source files of its body.

5 New constructs

5.1 Temporal window

The goal of this new construct is to provide an easy access to the past N values of a stream with a simple index and generate a good code for it (circular buffer). The window can be manipulated as an array of size N.

With temporal windows, **fby** delays can be replaced by a constant access to a window content:
fby `(e; N; i)` can be replaced by `w[N-1]` where `w` is a window of size $M \geq N$.

```
{
...
var w : T^M;
let
...
  w = window<<M>>(i^M)(e);
...
}
```

The size argument of the window specifies the number of past values it contains, the first parameter is the array defining the initial content of the window and and the second parameter is the flow observed through this window. Note that this initial array allows to do initializations that cannot be done with SCADE 6 **fby**.
Let `w = window<<N>>(I)(e)`, the window is defined by:

$$\begin{cases} \mathtt{w}[0] = \mathtt{I}[0]\text{->}\mathbf{pre}\ e \\ \forall k \in [1..N[,\ \mathtt{w}[k] = \mathtt{I}[k]\text{->}\mathbf{pre}\ \mathtt{w}[k-1] \end{cases}$$

Example:

```
node sliding_wa(x : float64) returns (wa : float64) {
var w : float64^N;
let
  w = window<<N>>(0.^N)(x);
  wa = (x + c[0]*w[0] + c[1]*w[1] + c[2]*w[2] + c[3]*w[3]) / d;
}
```

```
const
  N : int32 = 4;
  c : float64^N = [0.5, 0.25, 0.125, 0.0625];
  d : float64 = 1.0+c[0]+c[1]+c[2]+c[3];
```

This operator returns a sliding weighted average of the current value and the past four values. The current value of x is not in the window w and must be added separately.

The parameters of windows are given in two groups, which allows to define at once several windows of same size. Here is an example that computes the the sum of the last 10 values of two flows and returns the quotient of these two sums. The two flows are inilialized differently: one with zero and the other with one.

```
node sliding_q(x : float64; y : float64) returns (o : float64) {
var
  wx : float64^9;
  wy : float64^9;
let
  wx,wy = window<<9>>(0.^9, 1.^9)(x,y);
  o = (fold (+))<<9>>(x, wx) / (fold (+))<<9>>(y, wy);
}
```

Note in this example, the current value of x and y are integrated in the formula, thus the windows are of size 9 only and the sums integrate the current values.

5.2 Spacial window operator: pack

Operator **pack** is a spacial version of the sliding window that, given an array V, defines the array composed of successive, possibly overlapping, chunks of V. Let V be of size n and W defined by the equation

```
  W = pack<<k,m>>(V);
```

Thus W is an array of m chunks of size k and such that:

$$\forall i \in [0..m[, j \in [0..k[,\ \mathtt{W}[i][j] = \mathtt{V}[s*i+j]$$
$$\text{where } s = (n-k)/(m-1)$$

In this definition, s represents the stride used to move from one chunk to the next one.

To avoid ignoring the latest values in V because the division that defines s is not exact, the type check verifies the divisibility condition: $(m-1) \mid (n-k)$. This condition does not limit the expressiveness since it is always possible to satisfy it by taking a slice of V to get a smaller array that satisfies the condition which corresponds to explicitly select the ignored part of V.

To sum up the application conditions are:

$$m > 1 \quad \text{and} \quad k < n \quad \text{and} \quad (m-1) \mid (n-k)$$

Let $\mathtt{V} = [v_0, \dots, v_{n-1}]$ and $s = (n-k)/(m-1)$, then W is the following two dimensional array:

$$\begin{bmatrix} [& v_0 & , \cdots , & v_{k-1} &] \\ [& v_s & , \cdots , & v_{s+k-1} &] \\ & \vdots & , \cdots , & \vdots & \\ [& v_{(m-1)s}, & \cdots , & v_{(m-1)s+k-1} &] \end{bmatrix}$$

To visualize the operation with concrete values, let us take the case of $n = 10$, $k = 6$ and $m = 3$. One can easily verify that the application conditions are satisfied and that it leads to a stride $s = 2$. Applying a **pack** with these parameters on an array `V` $= [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, leads to taking the following chunks of size 6:

$$[0,1,2,3,4,5,6,7,8,9]$$
$$[0,1,2,3,4,5,6,7,8,9]$$
$$[0,1,2,3,4,5,6,7,8,9]$$

and have them all in an array of array, which gives:

$$\begin{bmatrix}[0,1,2,3,4,5]\\ [2,3,4,5,6,7]\\ [4,5,6,7,8,9]\end{bmatrix}$$

Note: to memorize the role of the static parameters, just remember that if `V` has type `T^n` then **pack**`<<k,m>>(V)` is of type `T^k^m`. The definition of the operation involves the size n of the array on which it applies and three other sizes, k, m and s all related by the equation $s = (n - k)/(m - 1)$; it has been chosen to parametrize **pack** by k and m because these dimensions will occur elsewhere in a model that uses the result of the operation while s is internal to the transformation since it does not appear in the type of the result.

Example: a 1-D Gaussian blur with a kernel of size 3 and padding.

```
function kernel (a:float64^3) returns (o:float64)
  o = (a[0] + 2*a[1] + a[2]) / 4;

function blur <<n>> (A:float64^n) returns (B:float64^n) {
  var C;
  let
    C = (map kernel) <<n-2>> (pack<<3,n-2>>(A));
    B = [C[0]] @ C @ [C[n-3]];
}
```

Note the size of the array resulting from the application of the kernel is smaller than the input array, the initial the padding is done by repeating the first and last cells (see definition of `B`).

Remark about code generation: the **pack** operator belongs to the same category as **reverse** and **transpose** in the sense it is often not necessary to construct its result to use it, in presence of a complete (i.e. with an index for each dimension) projection, introducing some index translation is enough.

Example: given a vector `A` of size $2N$, compute `o` defined by

$$\mathtt{o} = \sum_{i=0}^{N-1} (a_{2i} - a_{2i+1})$$

where `A` $= [a_0, a_1, \ldots, a_N, \ldots, a_{2N-1}]$.

Using **pack** it is easy to produce an array of pairs of successive coefficients, by definition of the operator, **pack**`<<2,N>>(A)` is the two-dimensional array:

$$[[a_0, a_1], [a_2, a_3], \ldots, [a_{2N-2}, a_{2N-1}]]$$

Then `o` can be computed by:

```
function root<<N>>(A:int32^(2*N)) returns (o:int32) {
var B : int32^2^N;
let
  B = pack <<2,N>>(A);
  o = (fold (function a, Bi => a+Bi[0]-Bi[1]))<<N>>(0,B);
}
```

Another solution is to first transpose the result of the pack that gives:

$$[[a_0, a_2, ...a_{2N-2}], [a_1, a_3, \ldots, a_{2N-1}]]$$

then map the subtraction point-wisely on the two items of this array and sum the content of the result:

```
function root<<N>>(A:int32^(2*N)) returns (o:int32) {
var B : int32^N^2;
let
  B = transpose (pack <<2,N>>(A));
  o = (fold (+))<<N>>(0, (map (function a, b => a - b))<<N>>(B[0],B[1]));
}
```

A third version that implements the re-arranged definition

$$\mathtt{o} = \left(\sum_{i=0}^{N-1} a_{2i}\right) - \left(\sum_{i=0}^{N-1} a_{2i+1}\right)$$

```
function root<<N>>(A:int32^(2*N)) returns (o:int32) {
var B : int32^N^2;
let
  B = transpose (pack<<2,N>>(A));
  o = (fold (+))<<N>>(0,B[0]) - (fold (+))<<N>>(0,B[1]);
}
```

Example: Convolutions on multidimensional data can be implemented by applying this transformation one dimension at a time. Below is an example of a pooling layer used in *Convolutional Neural Networks* (CNN). It consists basically in dividing by two the resolution of a 2-D image by taking neighborhoods of $2 * 2$ pixels of the original and produce one corresponding pixel in the resulting image that is given the max of the original pixel values:

```
const
  M : int16 = 100; -- nb lines
  N : int16 = 200; -- nb columns
  F : int16 = 5; -- features
  Ksz : int16 = 2; -- kernel size
  Kst : int16 = 2; -- stride
  Msub : int16 = (M - Ksz) / Kst + 1;
  Nsub : int16 = (N - Ksz) / Kst + 1;

function max(a: 'T; b: 'T) returns (m : 'T) where 'T numeric
  m = if a >= b then a else b;
```

```
function pool_max_kern <<Ksz>> (kernel: int8^Ksz^Ksz)
  returns (cell : int8)
  cell = max (max (kernel[0][0], kernel[0][1]),
              max (kernel[1][0], kernel[1][1]));

function pool_max_row <<M, Ksz, Msub>>(Col_pack: int8^M^Ksz)
  returns (Col: int8^Msub)
  Col = (map pool_max_kern<<Ksz>>)
          <<Msub>>(pack <<Ksz, Msub>>(transpose (Col_pack)));

function pool_max_array <<M, N, Ksz, Msub, Nsub>> (x: int8^M^N)
  returns (y: int8^Msub^Nsub)
  y = (map pool_max_row<<M, Ksz, Msub>>)<<Nsub>>(pack<<Ksz,Nsub>>(x));

function Pool_max <<M, N, F, Ksz, Kst>>(x: int8^M^N^F)
  returns (y: int8^((M-Ksz)/Kst+1)^((N-Ksz)/Kst+1)^F)
  y = (map pool_max_array<<M, N, Ksz, (M-Ksz)/Kst+1, (N-Ksz)/Kst+1>>)<<F>>(x);

function root (A : int8^M^N^F) returns (B: int8^Msub^Nsub^F)
  B = Pool_max<<M,N,F,Ksz,Kst>>(A);
```

5.3 Array flatten

flatten takes an array with at least two dimensions and returns the array corresponding to the flattening of these two dimensions into a single one. This operator has type: $\forall\tau.(\tau^m)^n \xrightarrow{0} \tau^{n*m}$.

let `B` = **flatten** `(A)` where `A` has a type of the form `T^M^N` and is such that `A[i][j]`=$a_{i,j}$; then `B` has type `T^(M*N)` and its content is defined by `B[k]`=$a_{(k\,div M),(k mod M)}$.

Example with `M`= 2 and `N`= 3:

if `A` $= [[a_{0,0}, a_{0,1}], [a_{1,0}, a_{1,1}], [a_{2,0}, a_{2,1}]]$
then **flatten**`(A)` $= [a_{0,0}, a_{0,1}, a_{1,0}, a_{1,1}, a_{2,0}, a_{2,1}]$

5.4 Tagged unions

Tagged unions allow the description of variant flows (i.e. flow that may have different type). Contrary to *union* types, the *tag* value ensures *disjointness* which means that the projection of the value carried by a flow v, at step t, is the same for any usage of the flow. The example below illustrates how a *tagged union* datatype can be defined:

```
type
  polar = {r: float32, t: float32};
  cart  = {x: float32, y: float32};
  coord2D =
      Polar     {polar}
    | Cartesian {cart};
```

From a syntactic point of view, the description of variant values can be seen as a generalization of structures:

```
type either3 =
    Either1 {}                  -- empty structure, i.e. no value
  | Either2 {int32}             -- unlabeled single-field structure
  | Either3 {a: bool, b: int32}; -- multiple fields structure
```

The following example shows how tagged unions can be constructed and used:

```
type
  polar = {r: float32, t: float32};
  cart  = {x: float32, y: float32};
  coord2D =
      Polar     {polar}
    | Cartesian {cart};

function arctan(x: float32) returns (y: float32);
function cos(x: float32) returns (y: float32);
function sin(x: float32) returns (y: float32);
function sqrt(x:float32) returns (y: float32);

function pow_2(x: float32) returns (y: float32)
  y = x * x;

function cartesian2polar(p: coord2D) returns (np: coord2D default = p)
  np: activate when p match
      | Cartesian {z}:
        np = Polar {r : sqrt(pow_2(z.x) + pow_2(z.y)), t : arctan(z.y / z.x)};
      | Polar _      :
        np = p;
  ;

function polar2cartesian(p: coord2D) returns (np: coord2D)
  np: activate when p match
      | Cartesian _ :
        np = p;
      | Polar {x}   :
        np = Cartesian {x : x.r * cos(x.t), y : x.r * sin(x.t)};
  ;
```

As introduced in the example, the wildcard symbol _ is used to denote anonymous, and therefore unused, variable.

From a language point of view, the clock of a variant is the tag itself. These values can be accessed using control-flow constructs as illustrated below:

```
function normalize(p: coord2D) returns (c: cart)
  c: activate when p match
     | Cartesian {cp}: c = cp;
     | Polar     {pp}: c = { x: pp.r * cos(pp.t), y: pp.r * sin(pp.t) };
  ;

function abscissa(p: coord2D) returns (x: float32)
  x = normalize(p).x;
```

In addition, the **when match** primitive is defined as a mean to access the value carried by a variant. Below is a dataflow definition of `cartesian2polar`, using **when match**:

```
function normalize_df(clock p: coord2D) returns (c: cart) {
var pp : polar when (p match Polar);
let
  pp = p when match Polar;
  c  = merge(p;
              p when match Cartesian;
              { x: pp.r * cos(pp.t), y: pp.r * sin(pp.t) });
}
```

The semantics of **when match** primitive relies on the same principle as **when**: it has no values if the tag of the *left* operand does not match the *right* operand.

Values of type tagged union can be used as a selector expression in a **case**, in this usage the pattern cannot capture the values carried by the variant. Here is an example of a simple predicate:

```
function is_polar (p: coord2D) returns (c: bool)
  c = (case p of
      | Polar : true
      | _     : false);
```

Note about variables in patterns: a pattern can contain a variable to capture the value encapsulated in a union. This can be seen in the examples, for instance variable `cp` in the function `normalize` above. There is a usage restriction on this kind of variables, they cannot be the object of a **last** i.e. writing **last** `'cp` is forbidden. The reason for this restriction is that allowing this access through a **last** could require an initialization of the memory at the variable introduction point which would make the syntax of pattern heavier for few interesting cases. If there is a need to use a last on this variable, one can introduce a new variable (open a **var** section) in the concerned branch, define it as equal to the one captured by the pattern and with the necessary memory initialization.

5.5 Anonymous operators

The goal of this extension is to allow the introduction of an operator at its application point by giving its definition and without giving it a name. This features exists in many programming languages and often called *lambda expression.* Since Swan is a first-order language this new form can only appear as an applied operator and not as any expression we call it *anonymous operator.*

The anonymous operators exist in two syntactic forms:

- one that follows the user operator declaration syntax with an explicit interface (inputs and outputs) and a body;
- one lighter where inputs are just named and their type is inferred, the outputs are defined by an expression, and a body with local declarations can be optionally introduced (Note: the input variables in this are supposed to be flows and not groups to allow inference).

Note: in the first form, the interface is similar to the one of a declared operator except that it cannot be polymorphic i.e. introduce type variables and constraints in its signature. The reason for this choice is that since the operator is instantiated once and where it is defined, it can use the types (including type variables) that are visible in the scope where it is introduced. In this condition, allowing polymorphism for this form would not improve expressiveness. The definition of the function `MatProd_1` below shows the use of a type variable in the interface of the anonymous operator, this type variable is the one introduced in the interface of `MatProd_1`.

The examples below illustrate these two syntax with an operator that computes a matrix product.
The matrix product with explicit interface in anonymous operator:

```
-- matrix product: A(m,n) * B(n,p)
function MatProd_1 <<m, n, p>> (A: 'T^m^n; B: 'T^n^p)
  returns (C:'T^m^p) where 'T numeric
  C = (map
        (function (u:'T^n) returns (w:'T^m)
           w = (map (function (v:'T^n) returns (x:'T)
                        x = (fold (+))<<n>>(0, (map (*))<<n>>(u, v));))
                <<m>>(transpose (A));))
        <<p>>(B);
```

The matrix product with light form of anonymous operator:

```
-- matrix product: A(m,n) * B(n,p)
function MatProd_2 <<m, n, p>> (A: 'T^m^n; B: 'T^n^p)
  returns (C:'T^m^p) where 'T numeric
  C = (map (function u =>
          (map (function v => (fold (+))<<n>>(0, (map (*))<<n>>(u, v))))
               <<m>>(transpose (A))))
        <<p>>(B);
```

The Cholesky matrix decomposition using both notations:

```
function sqrt(x : 'T) returns (y : 'T) where 'T float;
function sqrt32(x:float32) returns (y:float32) specialize sqrt;
function sqrt64(x:float64) returns (y:float64) specialize sqrt;

function ScalProdRow <<N>> (U: 'T^N^N; i:int32; j: int32)
  returns (p :'T) where 'T numeric {
var V : 'T^N;
let
  V = (mapi
        (function k =>
           (U.[i][k] default 0) * (U.[j][k] default 0)))<<N>>();
  p = (fold (+))<<N>>(0, V);
}

function Chol <<N>> (A: float64^N^N) returns (L: float64^N^N)
  L = (mapfoldi
        (function j, S =>
           (mapfoldi
             (function (i: int32; S: float64^N^N) returns (Sa: float64^N^N) {
              var C : float64;
              let
                Sa, C :
                    activate if i > j
                    then {
                    let
                      Sa = S;
                      C  = 0;
                    } else if i = j
                    then {
                    let
                      C = sqrt((A.[i][i] default 0) - ScalProdRow<<N>>(S, i, i));
                      Sa = (S with [i][i] = C);
                    } else { -- i < j
                    let
                      C = ((A.[j][i] default 0) - ScalProdRow<<N>>(S, i, j))
                          / (S.[i][i] default 0);
                      Sa = (S with [j][i] = C);
                    };
                })
              <<N>>(S)))
         <<N>>(0^N^N);
```

Main usage of anonymous operators:

- as an iterated operator, which avoids defining a named operator used only in that context, and allows pushing expressions inside the definition without using fancy vectorisation of expression to distribute over the iteration;
- as a local scope, to activate or reset a set of equations.

5.6 Operator partial application

The partial application allows to specialize some of the inputs of an operator instance by giving it a partial list of effective parameters. A simple example, let `f` be the operator defined by:

```
function f (a: int32; b: int32; c: int32) returns (o: int32)
  o = a * b + c;
```

one of its inputs can be specialized before the final application. For instance `(f \ _, 7, _)` represents the operator after specializing its argument `b`. This can be seen as a shortcut for the following anonymous function:

```
(function x, y => f(x, 7, y))
```

Partial application is useful in iterations where some arguments of the iterated operator are constants in the iteration; in SCADE 6 this is generally done by vectorizing these arguments. The example of the matrix product given below illustrate this combination with iteration: in `MatProd`, the definition is given by iterating a specialized instance of `MatVectProd`; the same principle is used in the definition of `MatVectProd` with a specialization of `ScalProd`.

```
function prod_sum (acc_in: 'T;  ui: 'T; vi: 'T) returns (acc_out:'T) where 'T numeric
  acc_out = acc_in + ui * vi;

-- scalar product of two vectors: u . v
function ScalProd <<n>> (u: 'T^n; v: 'T^n) returns (w:'T) where 'T numeric
```

```
    w = (fold prod_sum) <<n>> (0, u, v);

  -- product of a matrix by a vector: A(m,n) * u(n)
  function MatVectProd <<m, n>> (A: 'T^m^n; u: 'T^n)
    returns (w: 'T^m) where 'T numeric
    w = (map (ScalProd <<n>> \ _, u)) <<m>> (transpose (A));

  -- matrix product: A(m,n) * B(n,p)
  function MatProd <<m, n, p>> (A: 'T^m^n; B: 'T^n^p)
    returns (C:'T^m^p) where 'T numeric
    C = (map (MatVectProd <<m, n>> \ A, _)) <<p>> (B);
```

Some constraints are required because of the implicit character of the construct, `_` represents a position in the positional interpretation of the group of inputs. These constraints already exists for equation left-hand sides. A labelled version that works with named groups is supported. For instance the `MatProd` function could be written:

```
  -- matrix product: X(m,n) * Y(n,p)
  function MatProd <<m, n, p>> (X: 'T^m^n; Y: 'T^n^p)
    returns (C:'T^m^p) where 'T numeric
    Z = (map (MatVectProd <<m, n>> \ A:X)) <<p>>(Y);
```

In this version the formal parameters have been renamed `X, Y, Z` to distinguish these names from the one of `MatVectProd` formal parameter. Here the parameter `X` is explicitly mapped to `A` of the called function.

5.7 Forward: iterating on *space* with operations on *time*

Swan introduces a new kind of iteration that allows to consume and produce arrays as if they were a sequence in time. Here *time* refers to the (usual) synchronous cycle while *space* refers to arrays. This construct differs from higher-order iterators on two aspects:

- syntactically it is an expression with a form that is reminiscent of a for loop in an imperative language[2];
- semantically it does not correspond to a multi-instantiation of the operations it contains but to several steps forward of a single instantiation.

The interest of this iteration is that all the language features that facilitate the specification of sequential logic can be applied to arrays computation. For instance an automaton can be used to recognize a sequence of values composed of the n values contained in an array and considered in a raising index order.

The example below defines an operator `monotonic` that returns a Boolean stream which is true at cycle n if the input stream is monotonic i.e. increasing, decreasing or constant. Based on this operator, the predicate `sorted` that applies on an array and is true if the array is sorted (either increasing or decreasing order) can be defined by iterating `monotonic` on this array.

```
  -- true if the input flow is monotonic until the current cycle.
  node monotonic (a: int32) returns (o:bool default = true )
    o :
      automaton
        initial state S0:
        unless if (true) resume Constant;
```

[2]This form is actually inspired from loops in the functional parallel language SISAL (see SISAL on Wikipedia)

```
        state Constant:
        unless
          if (a > last 'a) resume Increasing;
          if (a < last 'a) resume Decreasing;
          -- otherwise a = last 'a and no transition is fired

        state Increasing:
        unless if (a < last 'a) resume NonMonotonic;

        state Decreasing:
        unless if (a > last 'a) resume NonMonotonic;

        state NonMonotonic:
          let o = false;
      ;

  -- true if the current array content is sorted (increasing or decreasing)
  function sorted(A: int32^5) returns (o:bool)
    o = forward <<5>> with [a]=A;
          let m = monotonic(a);
        returns (m);
```

The **forward** iteration makes the array `A` be treated as a finite flow of length 5 and returns the value taken by `monotonic` when treating the last element of `A`.
Note `sorted` is declared as a function, thus combinational, while it uses the node `monotonic` that is sequential; this aspect is discussed in 5.7.7.

Since the **forward** has a body that can contain any kind of equations, the node `monotonic` can be inlined and sorted written as follows:

```
  -- true if the current array content is sorted (increasing or decreasing)
  function sorted(A: int32^5) returns (o:bool)
    o = forward <<5>> with [a]=A;
          let m :
            automaton
              initial state S0:
              unless if (true) resume Constant;

              state Constant:
              unless
                if (a > last 'a) resume Increasing;
                if (a < last 'a) resume Decreasing;
                -- otherwise a = last 'a and no transition is fired

              state Increasing:
              unless if (a < last 'a) resume NonMonotonic;

              state Decreasing:
              unless if (a > last 'a) resume NonMonotonic;

              state NonMonotonic:
                let m = false;
            ;
        returns (m : default = true);
```

Here is another example that computes the sum of the elements of an input array `A` by iterating the operator `Integrator` with a **forward** iteration. `Integrator` defines the cumulative sum (or integration) of its input stream `x`.

```
node Integrator(x:int32) returns (y:int32 last=0) y = last'y + x;

function ArrayIntegrator(A:int32^5) returns (o:int32)
  o = forward <<5>> with [a]=A;
        let s = Integrator(a);
      returns (s);
```

Here again, the node `Integrator` can be inlined:

```
function ArrayIntegrator(A:int32^5) returns (o:int32)
  o = forward <<5>> with [a]=A;
        let s = last 's + a;
      returns (s : last = 0);
```

This example could also be defined with a **fold** iterator: `(`**fold** `(+))<<5>>(0, A)`. If we compare the two implementations, the one with the **fold** initializes the sum at the level of the instance while the one with **forward** allows to integrate it in the definition of the iterated node. The second form scales better when more than one accumulator is required, it avoids to have the list of accumulators in several (four actually) places: inputs and outputs of both definition and instance.

A very nice illustration of an array seen as a (finite) stream using Swan sequential primitives is the function `rotate_right` that given an array `A` produces an array with same content as `A` shifted to the right and which first element is the last element of `A`. The shifted is naturally implemented by the **pre** operator and setting the first element is done the same way as defining the first value or a stream. This leads to the following code:

```
function rotate_right <<n>> (A:int32^n) returns (B:int32^n)
  B = forward <<n>> with [a] = A;
        let b = A[n-1] pre a;
      returns ([b]);
```

This rotation is just a shift initialized with the last value; based on the same idea and using a very simple automata it is easy to express the insertion of a value in a sorted (increasing) array. An insertion can be decomposed in three steps: finding the insertion position, insert, shift the remaining values. To define it in term of forward, we can use a two-state automate where `Before` is the state before the insertion point and `After` is the state where the current element of the iteration are after the insertion point:

```
function insert <<n>> (x:int32; A:int32^n) returns (B:int32^n)
  B = forward <<n>> with [a] = A;
        let
          b : automaton
                initial state Before:
                unless if (x < a) resume After;
                  let b = a;
                state After:
                  let b = x pre a;
              ;
      returns ([b]);
```

Note: if the value `x` place in `B` (i.e. if it is smaller than `A[n-1]`) then the element `A[n-1]` will be present in B. Based on this function, we can define a node that collects for instance the five smallest values observed for a flow:

```
node five_smallest (x: int32) returns (X5s: int32^5 last = x^5)
  X5s = insert <<5>>(x, last 'X5s);
```

5.7.1 Understanding the syntax

Although close to a for loop in its form, the **forward** iteration has its own style to introduce its sub-parts to be more in line with the way it interacts with the sequential primitives of the language (**pre**, . . .). let us consider the following snippet:

```
forward <<n>> with <<i>> [a]=A; [b]=B;
  let
    ...
    c = ...;
returns ([c])
```

The first line reads : *forward* `n` *steps with* `i` *as a current index,* `a` *as a current element of* `A` *and* `b` *as a current element of* `B`*.*
This part uses a size expression `n` and introduces:

- size variable `i` with the current index, taking the successive values in range $[0..n-1]$.
- flow variables `a` and `b`.

The syntactic form `[a]=A` suggests that the array of elements `a` is defined by `A` and thus reads as *let* `a` *be the current element of* `A`*.*

The body of the forward contains flow definitions and in particular has to define the variables introduced by the return clauses.

Note: The syntax allows to name a return item of a forward: **forward** `<<n>>` . . . **returns** `(C=[c]);`. The associated feature (array recursive definition) is not yet defined. For this reason this aspect of the syntax is not commented yet.

5.7.2 Simple examples

Unidimensional simple filter (kernel of size 3), with padding:

```
function blur <<n>> (A:float64^n) returns (B:float64^n) {
  var C;
  let
    C = forward <<n-2>> with [a]=pack<<3,n-2>>(A);
          let ci = (a[0] + 2*a[1] + a[2]) / 4;
        returns ([ci]);
    B = [C[0]] @ C @ [C[n-3]];
}
```

Linear algebra operations:

```
function dot <<n>>(U:float64^n; V:float64^n) returns (d:float64)
  d = forward <<n>> with [u]=U; [v]=V;
        let s = last 's + u * v;
      returns (s : last = 0);

-- matrix * vector
function mat_vec <<n, m>>(A:float64^m^n; U:float64^m) returns (V:float64^n)
  V = forward <<n>> with [ai]=A;
        let vi = dot<<m>>(ai,U);
      returns ([vi]);

-- matrix * matrix
function mat_mat<<n,m,p>>(A:float64^m^n; B:float64^n^p) returns (C:float64^p^m)
  C = forward <<m>> with [ai] = transpose (A);
              <<p>> with [bj] = B;
        let cij = dot <<n>> (ai, bj);
      returns ([[cij]]);
```

The following examples are also linear algebra operations where the iterations are organized such that in the innermost one, there are no data dependency between the computation of the body for different indices. This kind of organization is called *vectorizable* and is well suited for a data-parallel execution:

```
-- vector * matrix (vectorizable)
function vec_mat<<n, m>>(U:float64^n; A:float64^m^n) returns (V:float64^m)
  V = forward <<n>> with [ai]=A; [ui]=U;
        let va =
          forward <<m>> with [aij]=ai; [lvj]=last'va;
            let vj = lvj + aij * ui;
          returns ([vj]);
      returns (va : last = 0^m);

-- matrix * matrix (vectorizable)
function mat_mat_v<<n,m,p>>(A:float64^m^n; B:float64^p^m) returns (C:float64^p^n)
  C = forward <<n>> with [ai] = A;
        let ci = vec_mat <<m, p>> (ai, B);
      returns ([ci]);
```

Sum of matrix elements: $\sum_{\substack{i\in[1..n]\\ j\in[1..m]}} a_{i,j}$

```
function elements_sum2D <<m, n>> (A:float64^m^n) returns (sum:float64)
  sum = forward <<n>>
                <<m>> with [[aij]]=A;
          let s = aij + last 's;
        returns (s : last = 0);
```

Let $U = (u_i)_{i\in[1..n]}$ and $V = (v_j)_{j\in[1..m]}$ the tensor product of these two vectors ($P = U \otimes V$) is the matrix $P = (p_{i,j})$ where $p_{i,j} = u_i \,.\, v_j$. The function below implements this definition.

```
function tensor_prod_vec_vec <<n,m>> (U:float64^n; V:float64^m)
```

```
    returns (P:float64^m^n)
    P = forward <<n>> with [ui]=U;
                <<m>> with [vj]=V;
          let pij=ui*vj;
        returns ([[pij]]);
```

Weighted sum of a 3D tensor elements:

```
  function elements_weighted_sum3D <<m, n, p>> (A:float64^m^n^p; W:float64^m^n)
    returns (sum:float64)
    sum = forward <<p>>
                  <<n>>
                  <<m>> with [[[a]]]=A; [[w]]=W;
            let s = w*a + last 's;
          returns (s : last = 0);
```

The example `elements_sum2D` above can be adapted to return the vector of all the partial sums i.e.:

$$\left(\sum_{\substack{i\in[1..k]\\ j\in[1..m]}} a_{i,j}\right)_{k\in[1..n]}$$

This is done by simply add square brackets in the return clause:

```
  function elements_partial_Sums2D <<m, n>> (A:float64^m^n)
  returns (Sums:float64^n)
    Sums = forward <<n>>
                   <<m>> with [[aij]]=A;
             let s = aij + last 's;
           returns ([s : last = 0]);
```

To help the reader understand what this function computes, let us apply it to the following matrix:

$$\begin{pmatrix} 1, & 2, & 3 \\ 4, & 5, & 6 \\ 7, & 8, & 9 \end{pmatrix}$$

it results in vector:

$$\begin{pmatrix} 6 \\ 21 \\ 45 \end{pmatrix}$$

This last example highlights the fact that the content of a forward with multiple dimensions is not reset between the dimensions, that is why it is easy to compute the sum of a multi-dimensional array elements. By default the streams defined by and inside a forward restart with the first value of their definition each time the forward is activated. Section 5.7.7 discusses this aspect with more details.

5.7.3 Using the current index

It is allowed to access the current index in the body of a **forward**. For instance the following example is a function returning an array filled with successive even values starting at 0:

```
function array_of_even <<N>> () returns (O:int32^N)
  O = forward <<N>> with <<i>>
        let oi = 2 * i;
      returns ([oi]);
```

In this example `i` appears in the computation and is not used to dereference arrays, in such a case it is convenient to use this index. In imperative languages (C, Ada, FORTRAN) and imperative algorithmic notation the loop index is generally used to dereference arrays and this is the only way to do so. For this reason it is common to think this way and the `dot` function defined in 5.7.2 may look familiar if written:

```
function dot <<n>>(U:float64^n; V:float64^n) returns (d:float64)
  d = forward <<n>> with <<i>>
        let s = last 's + U[i] * V[i];
      returns (s : last = 0);
```

instead of

```
function dot <<n>>(U:float64^n; V:float64^n) returns (d:float64)
  d = forward <<n>> with [ui]=U; [vi]=V;
        let s = last 's + ui * vi;
      returns (s : last = 0);
```

Having a close look at these two definitions, using a naming convention is actually enough to have a good matching with the mathematical formula ($U\,.\,V \;=\; \sum_{i=0}^{n-1} u_i.v_i$ where $U = (u_0, ..., u_{n-1})$ and $V = (v_0, ..., v_{n-1})$). From the type checking point of view, the first definition introduces the verification that the projection index `i` which is known to be in the range $[0..n[$ is in the bounds of the arrays `U` and `V` while in the second version the correctness is only based on the equality of the sizes of arrays `U` and `V` to n. These two implementations are equivalent but it is easier to reason on the second and thus it has a better potential for optimization.

Most of the time it is possible to avoid dereferencing, for instance the function `shifted_sum` defined by:

```
function shifted_sum <<n>>(A:float64^n; B:float64^n) returns (O:float64^(n-1))
  O = forward <<n-1>> with <<i>>
        let oi = A[i] + B[i+1];
      returns ([oi]);
```

can be written without projection:

```
function shifted_sum <<n>>(A:float64^n; B:float64^n) returns (O:float64^(n-1))
  O = forward <<n-1>> with [ai]=A[0 .. n-2]; [bsi]=B[1 .. n-1];
        let oi = ai + bsi;
      returns ([oi]);
```

where `bsi` means element of `B` at index successor of `i`.
Another example, the function `rev_sum` defined by:

```
function rev_sum <<n>>(A:float64^n; B:float64^n) returns (O:float64^n)
  O = forward <<n>> with <<i>>
        let oi = A[i] + B[n-i-1];
      returns ([oi]);
```

can also be written without projection:

```
function rev_sum <<n>>(A:float64^n; B:float64^n) returns (O:float64^n)
  O = forward <<n>> with [ai]=A; [rbi]=reverse(B);
        let oi = ai + rbi;
      returns ([oi]);
```

Note that in the second version the intention of considering `B` in reverse order is explicit (and clear) while in the first it is the result of an interpretation of the access `B[n-i-1]` and the knowledge that `B` is of size `n`.

Let us consider the example of matrix product and implement it using indices and projection to adopt a C-like style:

```
function mat_mat<<n,m,p>>(A:float64^m^n; B:float64^n^p) returns (C:float64^p^m)
  C = forward <<m>> with <<i>>
              <<p>> with <<j>>
        let cij = forward <<n>> with <<k>>
                    let c = last 'c + A[k][i] * B[j][k];
                  returns (c : last = 0);
      returns ([[cij]]);
```

In this case, with multidimensional arrays the primitive that will allow avoiding references is the transpose that allows to see array `A` with permuted dimensions:

```
function mat_mat<<n,m,p>>(A:float64^m^n; B:float64^n^p) returns (C:float64^p^m)
  C = forward <<m>> with [a_i] = transpose(A);
              <<p>> with [bj_] = B;
        let cij = forward <<n>> with [aki]=a_i; [bjk]=bj_;
                    let c = last 'c + aki * bjk;
                  returns (c : last = 0);
      returns ([[cij]]);
```

Even if the use indices and projection in the expressions make the text closer to an equivalent C-code, Swan does not allow projections on the equation left-hand side. The reason is that an equation in Swan guarantee to have a complete definition of the left-hand side; an array is not defined as the result of successive assignments of its cells while in C it is.

The `blur` function, pressented in 5.7.2 without capturing the index can also be written:

```
function blur <<n>> (A:float64^n) returns (B:float64^n) {
  var C;
  let
    C = forward <<n-2>> with <<i>>
          let ci = (A[i] + 2*A[i+1] + A[i+2]) / 4;
        returns ([ci]);
    B = [C[0]] @ C @ [C[n-3]];
}
```

which may look easier to understand than:

```
function blur <<n>> (A:float64^n) returns (B:float64^n) {
  var C;
  let
    C = forward <<n-2>> with [a]=pack<<3,n-2>>(A);
          let ci = (a[0] + 2*a[1] + a[2]) / 4;
```

```
            returns ([ci]);
        B = [C[0]] @ C @ [C[n-3]];
    }
```

because of the presence of **pack**; actually the version using **pack** gives more guarantees than the first one. This is due to the relation satisfied by the size of the array it applies on and its size parameters (`<<3,n-2>>`) given in 5.2. This relation ensures that all the items in `A` were considered while the first version ensures that the accesses are correct but not that all the elements were involved in the computation. Note that this function can also be defined in a quite compact way with a **pack** and a**map** applied to an anonymous operator that defines the kernel:

```
function blur <<n>> (A:float64^n) returns (B:float64^n) {
  var C;
  let
    C = (map (function a => (a[0] + 2*a[1] + a[2]) / 4)) <<n-2>> (pack<<3,n-2>>(A));
    B = [C[0]] @ C @ [C[n-3]];
}
```

the use of **map** avoids the introduction of `ci` present in the **forward** based definitions.

As a summary indices are accessible and their range is static, thus they can be used for a statically verified array projection (i.e. without default). Most of the time it is possible to reformulate a forward to avoid these projection without sacrifying the readability.

Note about static verification: The **forward** current indices have a statically known range and can be used in verified projection however this range verification is not yet specified.

5.7.4 Usecase of forward without current element

In the syntax, the **with** part which serves to capture current indices or current array elements is optional which may look weird when comparing **forward** with an imperative for-loop. Because the **forward** implicitly embeds a state since SWAN expressions are sequential, it is not always necessary to have a current element to define something useful. For instance the function that given a size `N` and an integer `K` returns the array of size `N` filled with successive integers starting at `K` can be written:

```
function fill_from <<N>>(K : int32) returns (O : int32^N)
  O = forward <<N>>
        let oi = K pre (oi + 1);
      returns ([oi]);
```

The body of the forward can be replaced by a node:

```
function fill_from <<N>>(K : int32) returns (O : int32^N)
  O = forward <<N>>
        let oi = count_from(K);
      returns ([oi]);

node count_from (K : int32) returns (cpt : int32)
  cpt = K pre (cpt + 1);
```

In this form, `fill_from <<N>>(K)` can be explained as the function that returns the `N` first values of the stream defined by `count_from(K)`.
For instance given the node `Fibonacci` defined by:

```
node Fibonacci() returns (f : int32) {
var w;
let
  w = window<<2>>([1, 1])(w[0]+w[1]);
  f = w[1];
}
```

the expression **forward** `<<10>>` **let** `f = Fibonacci()` **returns** `([f])` defines a constant flow of arrays containing the 10 first values of the Fibonacci sequence, i.e. the following value for ever:

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]\ ,\cdots$$

This degenerate case of the construct gives a simple pattern to define an array with the first values of a stream defined by a node.

5.7.5 Usecase of forward without arrays

As for iterators we presented forward as a way to traverse arrays to define new ones; and like the **fold** iterator it admits the degenerate case where no array is involved (`(`**fold** `f)` where `f` takes one accumulator and returns one accumulator). This case is not the most frequent usage but still relevant, in particular it allows to express iterative algorithms on values.

We illustrate this usecase with the example of a fast exponentiation algorithm (aka. *exponentiation by squaring*). This algorithm aims at computing x^p in the case where p is integer with a number of multiplications done that is at most $2.\log_2(p)$ multiplications instead of the $(p-1)$ of a naive implementation. This is based on the binary representation of p:

$$p \;=\; \sum_n b_n\,.\,2^n$$

that leads to:

$$x^p \;=\; \prod_n x^{b_n.2^n}$$

The x^{2^n} can efficiently be computed by successive squaring of x. b_n is in $\{0,1\}$, integrating its contribution is just an analysis of these two cases. This can be expressed by the three sequences (u_n), (v_n) and (w_n) defined by:

$$\begin{array}{lll} u_0 = p & v_0 = x & w_0 = \begin{cases} \text{if } p \text{ is odd} \\ \text{then } x \\ \text{else } 1 \end{cases} \\ u_n = u_{n-1}/2 & v_n = v_{n-1}.v_{n-1} & w_n = \begin{cases} \text{if } u_n \text{ is odd} \\ \text{then } w_{n-1} * v_n \\ \text{else } w_{n-1} \end{cases} \end{array}$$

Based on these sequences, we have b_n, the n^{th} bit of p in its binary representation that is equal to $u_n \bmod 2$ and that for all integer $i > \log_2(p)$, $u_i = 0$, thus $b_i = 0$ too, and finally $w_i = x^p$. Since SWAN allows to easily define sequences, let us define a function that takes a numeric value and an unsigned integer n of 8 bits. The type of n informs that the result can be reached in, at most, 8 iterations. It can be defined by:

```
function fast_exp (x: 'T; p: uint8) returns (x_pow_p: 'T) where 'T numeric
  x_pow_p =
    forward <<8>>
      var u; v;
      let
        u = p pre (u / 2);
        v = x pre (v * v);
        w = if u mod 2 = 1 then last 'w * v else last 'w;
    returns (w : last = 1);
```

Compared to the mathematical expressions of the sequences, (u_n) and (v_n) have been directly encoded with an initialized **pre** while (w_n) uses the fact that the base and the recurrence relation share the same expression provided **last** `'w` is initialized with `1`. Making these recurrence relations go forward for `8` steps give (w_8) that is equal to x^p.

Note: This example can be expressed with a **fold** using a multi-accumulator with all the **forward** internal state in the input and output accumulators. See section 6.15 for an alternative definition of `fast_exp` using a **fold**.

The first (and main) intuition of **forward** followed the idea that sequential operations, usually mapped on the idea of *time*, are, in the body of the construct, mapped to *space* represented as arrays (see 5.7). In the present case (i.e. without array) we can use another intuition, the one that **forward** introduces a local clock that is faster than the cycle and all the sequential logic contained in its body executes faster than the outside of the loop. [3]

The important thing is that both intuitions are helpful and coherent with each other. However the **forward** as it is proposed in Swan as an array iteration. For this reason it is preferably presented here à la SISAL.

5.7.6 Typing and scoping in forward expressions

This section aims at giving an informal presentation of the type consistency of the **forward** construct.

Scope₀

forward

Scope₁ $<< n_1 >>$ **with** …

…

$<< n_k >>$ **with** $<< i >>$ $\underbrace{[\cdots[}_{m}a]\cdots]$=$A$; …

$<< n_{k+1} >>$ **with** …

…

unless *cond1*

Scope₂ -- *forward body*

…

until *cond2*

returns ($\underbrace{[\cdots[}_{p}b$: **last** $= c$ **default** $= d\,]\cdots]$)

The scopes are the following:

[3] This is typically the point of view proposed by Cédric Pasteur in his PhD dissertation, see Time Refinement in a Functional Synchronous Language.

- *Scope₀* is the scope that contains the whole **forward** expression;
- *Scope₁* is the scope that introduces the current elements of the iteration;
- *Scope₂* is the scope of the **forward** body.
- These scopes are nested as follows: *Scope₂* is in *Scope₁* and *Scope₁* is in *Scope₀*.

The expression scopes are:

- expressions in **with** clauses $(A, \ldots)$ are in *Scope₀*;
- expressions for **last** and **default** in **returns** clauses (c and d) are in *Scope₀*;
- *cond1* is in *Scope₁*;
- *cond2* is in *Scope₂*.

The scopes in which the variables are introduced are:

- a and index i in *Scope₁*;
- b in *Scope₂*.

The type constraints are:

- all the n_i are size expressions;
- *cond1* and *cond2* are Booleans;
- a has type a τ_f (in particular a is a flow);
- i has type **integer** and considered as a size;
- $0 \ < \ m \ \leq \ k$;
- A has type ${\tau_f}^{n_k \cdots n_q}$ where $q = k - m + 1$;
- b, c and d have the same type ${\tau_f}'$;
- the returned flow is of type ${\tau_f}'$ if $p = 0$ otherwise (i.e. if $p > 0$) it is an array of type ${{\tau_f}'}^{n_p \cdots n_1}$.

The generic form used here to specify the rules returns a single flow, the same rules apply component-wise for the case of a more complex returned group. The specification of the returned group is given in section **??** about group checking.

Note: The variables introduced by the return of a forward can have a last and a default value specifies; the expressions that define these values are in the same scope as the one of the forward. This differs from the other local declarations for which both the variable and the last/default expressions are in the same scope. Moreover the default value specified in the return clause holds inside the forward and out of the forward in the case of a partial iteration. The snippet of code and its equivalent version below shows this interpretation:

```
let
  X, y =
    forward <<N>> with [a]=A; ...
    let
      -- body of the forward
    until c
    returns ([u : last = d1 default = d2], v : last = e1 default = e2);
```

where e1, e2, d1 and d2 are arbitrary expressions. is equivalent to:

```
var ld1; ld2; le1; le2;
let
  ld1 = d1;
  ld2 = d2;
  le1 = e1;
  le2 = e2;
  X, y =
    forward <<N>> with [a]=A; ...
    var
      u default = ld2 last = ld1;
      v default = le2 last = le1;
    let
      lu = u;
      lv = v;
      -- body of the forward
    until c
    returns ([lu : default = ld2], lv : default = le2);
```

In this equivalent version the last declarations have moved to the local declarations of the forward while the default ones have been duplicated, one for the local definitions and one for the returned values that are needed in the case of a partial iteration.

5.7.7 Restart and resume forward

In the previous sub-sections, **forward** has been used without expliciting its behaviour wrt. the sequential operator it contains. We also have seen that by default, the keyword **forward** used alone means that the content of the whole construct is reset each time it is activated (i.e. on its clock). Because of this reset the resulting behavior is combinational on flows, i.e. the sequential logic it contains is used for arrays treatment (in space), but does not refer to the previous synchronous cycle (in time). It is possible to make this default more explicit in the source of the model by using the keyword **restart**:

```
... forward restart <<N>> with ...
```

which is perfectly equivalent to:

```
... forward <<N>> with ...
```

But when putting a **resume** we get something different that is not combinational anymore and the state of the body at the end of the iteration is the one that will be used for the first iteration of the next activation.

For example, consider a signal s that is sampled with a sample time n times smaller than the one of the application that uses it. This means that each time the application is activated, it takes as input an array with n samples of s. Now if the application needs to integrate this signal, we have to cumulate all the values in the array sample with all the one from the previous steps. We can define the integrator `integr_higher_res` that is able to integrate a signal with a finer temporal resolution of a factor `n`:

```
node integr_higher_res <<n>> (S_array: float64^n) returns (o: float64)
  o = forward resume <<n>> with [s]=S_array;
        let sum = last 'sum + s;
      returns (sum : last = 0.0);
```

Here is another simple illustration of what can be done with the **resume** form. Consider the operator `Fibonacci` defined in 5.7.4 that produces the Fibonacci sequence; the expression

```
      forward resume<<3>> let f = Fibonacci() returns ([f])
```

produces the following flow:

$$[1, 1, 2], [3, 5, 8], [13, 21, 34] , \cdots$$

This time the flow is not constant as it is the case in the example of section 5.7.4. This is because the process under the **forward** is resumed and then starts a new cycle where it terminates the previous one. The Fibonacci sequence is here produced by slices of 3 values in an array.

The introduction of **resume** and **restart** allows to express the unfolding of a several dimensions **forward** in an equivalent nesting of single dimension ones in the following way:

```
forward <<n>> with [a]=A;
        <<m>> with [[b]]=B;
  -- ... body ...
returns (o: last = v)
```

can be rewritten:

```
forward restart <<n>> with [a]=A; [Bi]=B;
  let local_o =
    forward resume <<m>> with [b]=Bi;
      -- ... body ...
    returns (o: last = v);
returns (local_o)
```

We can see that multi-dimensional iteration is an interesting shortcut that avoids introducing some variables (here `Bi` and `local_o`) and having to properly set the resume. Here is an example of this transformation applied to function `elements_sum2D` defined in 5.7.2:

```
function elements_sum2D <<m, n>> (A:float64^m^n) returns (sum:float64)
  sum = forward <<n>> with [Ai]=A;
          let local_s =
            forward resume <<m>> with [aij]=Ai;
              let s = aij + last 's;
            returns (s : last = 0);
        returns (local_s);
```

Using a **resume** allows to simplify the management of the state variables, for instance in the example `sumStrictLowerTriangle` in section 5.7.8, each **forward** introduced its state to cumulate the sum and the nested one was initialized with the top one. By introducing and initializing the state only in the nested one and by resuming this state, it can be defined equivalently by:

```
function sumStrictLowerTriangle <<N>> (A: 't^N^N)
  returns (o: 't) where 't numeric
  o = forward restart <<N>> with <<i>> [Ai]=A;
        let _, sum =
          forward resume <<N>> with <<j>> [aij]=Ai;
          unless j >= i
            let s = last 's + aij;
          returns (s: last = default = 0);
      returns (sum);
```

Here is an example that shows how a multi-dimensions iteration allows to easily introduce alternance of **resume** and **restart** iteration. Let us write the expression that computes the following sum of products:

$$\sum_{\substack{i\in[1..n]\\ j\in[1..m]}} \left(\prod_{k\in[1..p]} a_{i,j,k} \right)$$

the following expression does it:

```
forward <<n>>
        <<m>> with [[aij_]]=A;
  let sum_ij = last 'sum_ij +
    forward <<p>> with [aijk]=aij_;
      prod_ijk = last 'p * aijk;
    returns (prod_ijk : last = 1);
returns (sum_ij : last = 0);
```

With function `sumStrictLowerTriangle` we have seen that by default a forward is **restart** and that in the multi-dimensional case, these dimensions can be made explicit with only mono-dimensional iterations where the top level one is **restart** and the other **resume**. So, in this example, semantically we have the following: **restart** for the top dimension (size `n`), **resume** for the second dimension (size `m`) and **restart** for the latest dimension (size `p`):

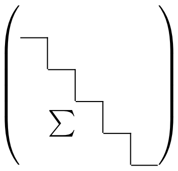
```
forward restart <<n>> with [ai_]=A
  let sum_ij =
    forward resume <<m>> with [aij_]=ai_;
      let sij = last 'sij +
        forward restart <<p>> with [aijk]=aij_;
          prod_ijk = last 'p * aijk;
        returns (prod_ijk : last = 1);
    returns (sij : last = 0);
returns (sum_ij);
```

5.7.8 Partial forward iteration

In the case of a single dimension forward, it is possible to stop the iteration either by a condition that is evaluated before the activation of the body, introduced by **unless**, after this activation, introduced by **until** or both.

When only one exit condition is specified, the group defined by the **forward** is implicitly extended in its first position with an integer flow that gives the iteration index. When two exit conditions (one **unless** and one **until**) are specified, the defined group is extended with two flows in that order: the iteration index and a Boolean information. The precise semantics of these implicit flows are given later in this section.

The following example illustrates the use of a guard before the body of the forward introduced by the keyword **unless**. The iteration is then limited by a static bound (here N) and a condition that makes leave the body as soon as it is ready. Given an integer N and a matrix $N \times N$, the goal is to compute $\sum_{i>j} a_{ij}$ which visually corresponds to:



```
-- sum of the array elements that are strictly below the diagonal
function sumStrictLowerTriangle <<N>> (A: 't^N^N)
  returns (o: 't) where 't numeric
  o = forward <<N>> with <<i>> [Ai]=A;
        let _, sum =
          forward <<N>> with <<j>> [aij]=Ai;
          unless j >= i
            let s = last 's + aij;
          returns (s: last = default = last 'sum);
      returns (sum: last = 0);
```

The returned index is ignored here (underscore in first position of the left hand side).
To illustrate partial iteration mixed with a resume, let us consider the example of the high resolution integrator introduced earlier in 5.7.7 in a context where the application is not executed with a rate that has a constant multiple of the signal sample time. This time, `k` is the number of samples to be considered for the integration and it varies, with a maximum value of `n`. With this new requirement the operator `integr_higher_res` becomes:

```
node integr_higher_res <<n>> (k: int32; S_array: float64^n) returns (o: float64)
  _, o = forward resume <<n>> with <<i>> [s]=S_array;
      unless i >= k
        let sum = last 'sum + s;
      returns (sum : last = 0.0);
```

Now let us consider the general form of a partial iteration:

```
k, out_c, xx, yy =
  forward <<N>> with ...
  unless c
    var ...
    let
      x = def_x;
      y = def_y;
      ...
  until d
  returns ( x : last = xl default = xd,
           [y : last = yl default = yd])
```

where xx is a value produced by the last iteration and yy is an array which elements are defined one by iteration. Let us first focus on the definition of k and out_c, the three stop cases to define for the iteration are:

- the forward is not stopped by any condition (*endfwd*)
- it is stopped by condition c (*unless*)
- it is stopped by condition d (*until*)

The output index and condition are defined depending on these three case:

	index k	condition out_c	number of actual steps
endfwd	= N	**false**	N
unless	< N	**false**	k
until	< N	**true**	$k + 1$

The last column of the table gives the number of times the body was activated i.e. the effective number of steps forward. When only one exit condition is specified, the column "index" still holds.
For the other productions of the expression (here x and [y]) the output x takes the default value xd if the body is not activated (i.e. if c is true for the first index) and [y] returns the array with the elements defined by the iterations of the body completed with the default value yd. (see 5.7.9 for a more precise definition).

A simple example of a find function that returns the index k of a value b if it is present in an array A of size N and a Boolean that gives the presence information:

```
function find <<N>> (b: 'T; A: 'T^N) returns (found: bool; k: int32) {
  let
    k = forward <<N>> with [a]=A;
        unless a = b
        returns ();
    found = k < N;
}
```

Note: this example can be written equivalently with **until** a = b.

Here is a find function that returns the index of the first value in an array of integers A that divides a given integer n.

```
function findDivisor <<N>> (n: int32; A: int32^N)
  returns (i: int32; found : bool; o: int32) {
let
  i, o =
    forward <<N>> with [a]=A;
      var divides;
      let
        divides = n mod a = 0;
        divisor = if divides then a else last 'divisor;
    until divides
    returns (divisor : last = 0);
  found = i <> N;
}
```

In this function, the value of output `o` only makes sense when the Boolean `found` is **true**.

Introducing a union type (see 5.4) to handle the case where no element satisfying the condition allows to correlate the value found and the fact it was found. This is the purpose of the second implementation below:

```
type int32_option = None {} | Some {int32};

function findDivisor_2 <<N>> (n: int32; A: int32^N)
  returns (i: int32; o: int32_option)
  i, o =
    forward <<N>> with [a]=A;
      var divides;
      let
        divides = n mod a = 0;
        divisor = if divides then Some{a} else last 'divisor;
    until divides
    returns (divisor : last = None);
```

In the case `A` does not contain any divisor of `n`, the returned index value is N and the value is `None` .

To illustrate the mix of **unless** and **until** conditions, the function `findTheCulprit` that searches the position (line `l` and column `k`) of the element that make the sum exceed a ceil `C`.

```
function findTheCulprit <<N>> (A: int32^N^N; C: int32)
  returns (l: int32; k: int32)
    l, k =
      forward <<N>> with <<i>> [Ai]=A;
        var found;
        let local_k, found =
          forward resume <<N>> with <<j>> [aij]=Ai;
          unless j >= i
            var sum last = 0;
            let sum = last 'sum + aij;
          until sum > C
          returns ();
      until found
      returns (local_k);
```

Static rules about default values. When using a partial forward, the size of the computed arrays (those in the return) is the size specified by the forward, regardless of the stop condition; it is thus needed to have a default value specified for the array outputs. For the accumulator ones it is a bit different since as soon as one step inside the forward is done, the output has a defined value and having an **until** exit only guarantee to have at least one effective step done. A default value is thus required for an accumulator only in presence of an **unless** condition. The following table resumes this; a box check in the cell meaning that a default value must be provided:

	accumulator	array
unless	✓	✓
until		✓

For instance, in the example `findDivisor` the variable `divisor` has no default specified since it is a simple accumulator and the exit condition is of kind **until**.

5.7.9 Forward semantics sketch

Let us define the invariants that characterize a forward expression, we first consider the case by default (i.e. **restart**), using the rather general form:

```
k, out_c, xx, yy =
  forward <<N>> with ...
  unless c
    var ...
    let
      x = def_x;
      y = def_y;
      ...
  until d
  returns ( x : last = xl default = xd,
           [y : last = yl default = yd])
```

As `k` is the output index i.e. in the range [1..`N`], the definition of the invariants can be split in three cases: the iterations that are before `k` and after `k` where `k` < `N` . When `k` is smaller than `N`, the Boolean expression $c[k] \vee d[k]$.

1. before `k`

$$\forall i \in [0..k[. \left(\begin{array}{ll} & \neg c[i] \\ \wedge & \neg d[i] \\ \wedge & x[i] = T_{ime}toS_{pace}(\texttt{def\_x}, i) \\ \wedge & y[i] = T_{ime}toS_{pace}(\texttt{def\_y}, i) \\ \wedge & \ldots \end{array} \right)$$

2. after `k` when `k<N`

$$\bigwedge \begin{array}{lll} c[k] & \Rightarrow & \forall i \in [k..N[. \left(\begin{array}{ll} & x[i] = \textbf{if } \texttt{k=0} \textbf{ then } \texttt{xd} \textbf{ else } x[i-1] \\ \wedge & y[i] = \texttt{yd} \end{array} \right) \\ \neg c[k] \wedge d[k] & \Rightarrow & \left(\begin{array}{ll} & x[k] = \texttt{def\_x}[k] \wedge y[k] = T_{ime}toS_{pace}(\texttt{def\_y}, k) \wedge \ldots \\ \wedge & \forall i \in]k..N[. y[i] = \texttt{yd} \end{array} \right) \end{array}$$

3. when `k=N` the definitions in 1 hold.

Where the function $T_{ime}toS_{pace}$(e, i) gives the array defined as a the prefix of flow e at index i. Given the index (defined by previous equations), the other produced flow definitions are:

$$\begin{array}{ll} \texttt{out\_c} & = \neg c[k] \wedge d[k] \\ \texttt{xx} & = x[k] \\ \texttt{yy} & = y \end{array}$$

If there is only an **unless** (resp. **until**) the equations hold considering $\forall i. \neg c[i]$ (resp. $\forall i. \neg d[i]$).

Synchronization of last and default : it is allowed to share the expression used for the last and the default in a return clause with the following notation: `x : `**last**` = `**default**` = e`. This is semantically equivalent to `x : `**last**` = e `**default**` = `**last**` 'x`.

TODO: all *restart/resume,* $T_{ime}toS_{pace}()$*, ...* ⇐

5.7.10 Example: a vote algorithm

The problem we want to solve here is the one of a vote, let us consider an array of `Nb_votes` votes that are represented as values of a given type `T` for which we only need to have the comparison operation [4]. The goal is to find the value present in strictly more than half of the total votes if such a value exists and warns when there is no majority.

Boyer and Moore proposed in the the 80's a very nice algorithm that solves the question with a constant (i.e. independant of `Nb_votes`) amount of memory and a linear time complexity. The principle is actually favorable to Swan as it is based on a sequential treatment of the input values i.e. it considers the votes in the array order and does not need to do random access in the array of votes. The algorithm, as it is given in its Wikipedia page is:

```
- Initialize an element m and a counter c with c = 0
- For each element x of the input sequence:
    - If c = 0, then assign m = x and c = 1
    - else if m = x, then assign c = c + 1
    - else assign c = c - 1
- Return m
```

the computed value `m` is a candidate for the majority winner in the sense that if a majority exits it is for `m`. The principle is to treat one vote at a time and update the state defined by the two variables `c` and `m`. Here is the definition of node `vote` that represents the three inner bullets and the state:

```
node vote (x: 'T) returns (m: 'T last = 0) where 'T numeric {
  var c: int32 last = 0;
  let
    c, m : activate if last 'c = 0
           then { let m = x; c = 1; }
           elsif m = x
           then c = last 'c + 1;
           else c = last 'c - 1;
         ;
}
```

The two formulations are very close, the access to the past value of a variable (i.e. part of the state) is done through a special construct **last**. `vote` is the stream function proposed by the algorithm, now it just needs to be applied to the array of votes and this is exactly what **forward** does; expression
forward `<<Nb_votes>>` **with** `[x]=Votes;` **let** `c = vote(x);` **returns** `(c)`
takes an array of votes and returns a candidate for majority. It is then necessary to check that the found candidate has effectively the majority, this is simply done by a second pass on the array, counting the number of occurrences of the candidate:

```
function count_votes <<N>> (x: 'T; A: 'T^N) returns (n: int32)
  n = forward <<N>> with [a]=A;
        let s = if a = x then last 's + 1 else last 's;
      returns (s : last = 0);

const Nb_votes : int32 = 10;

node root (Votes : int8^Nb_votes) returns (candidate: int8; has_majority: bool) {
```

[4]note that this is always the case in Swan, the comparison operations are defined for any type

```
    let
      candidate =
        forward <<Nb_votes>> with [x]=Votes;
          let c = vote(x);
        returns (c);
      has_majority = 2 * count_votes <<Nb_votes>> (candidate, Votes) > Nb_votes;
  }
```

An alternative implementation in SWAN consists in using a simple two states automaton which allows to not having to give an arbitrary initial value for **last** `'m` since the structure of the automaton guarantees that `m` is always defined:

```
  node vote (x: 'T) returns (m: 'T){
    var c; -- no need to have an initial value, the structure of the automaton
           -- guarantees that last 'c is not read at the first cycle
    let
      c, m : automaton
             initial state NoCandidate:
               let c = 1_i32; m=x;
             until resume Candidate;
             state Candidate:
               let c = if m = x then last 'c + 1 else last 'c - 1;
             until if (c = 0) resume NoCandidate;
           ;
  }
```

Anyway, from a pure performance point of view, we can anticipate that the first one will give better code, without additional state memory for the automaton and additional code for the automata machinery. For such a simple case where the state of the automatons deduces directly from the value of the state variable `c` it is not the best idea to introduce an automaton, but it is interesting to consider this alternative.

In practice, it is an interesting problem for embedded systems as they often use voting mechanisms to select a value among a redundant set of noisy and possibly faulty sources supposed to observe the same flow.

5.8 Diagrams

Extends operators and scopes with the possibility to describe the topology of a block diagram without having to encode it as a set of equations. A diagram is a list of object instances identified by a *luid* (for *locally unique id*) which must be unique within the operator that contains it.

5.8.1 Blocks, expressions and definitions

Here is an example of an operator defined by a block diagram:

```
  node integrator (i: float64) returns (o: float64) {
    diagram
      (#1 $adder_1 block (function a, b => a + b))
      (#2 $delay_1 expr pre #20
        where (#20 group))
      (#3 $init_1 expr 0.0 -> #30
        where (#30 group))
      (#4 expr i)
      (#5 def o)
```

```
        (wire #2 => #30)
        (wire #1 => #20, #5)
        (wire #4 => #1.(1))
        (wire #3 => #1.(2))
    }
```

5.8.2 Wires

A wire object is defined by a list of connections that are either ports or the empty connection denoted `()`. Wires are oriented in the sense that the first element of the list is considered as the source, i.e. the one that defines the flow carried by the wire, and the destinations that consume this flow. The general form is:
`(` **wire** *<source>* => *<dest_1>*, . . . , *<dest_n>*)
The empty connection corresponds to a wire that has some unconnected termination. This connection can be used either in source or target position.

Associated checks are:

- a warning is emitted if a destination is empty;
- an error is emitted if the source is empty and at least one destination is not empty.

5.8.3 Diagram specific checks

Objects:

Rule	Rationale
unicity of LUID in the scope of the operator	
expr and **def** objects shall have a LUID	necessary to be connected.
block objects shall have either a LUID or an operator instance with an INST_ID	necessary to be connected.
all the ports connected by a wire shall have a compatible group type	
all the input flows of a block must be connected at most once	
all the output flows of a block must be connected at most once	avoid having multiple consumption hidden in an output group, shall be made explicit by spliting the output wire.
all the ports connected by a wire must reference instances present in the same diagram	a wire is a graphical object that belongs to a single diagram; this diagram contains its source and destinations. This rule enforces the use of a named flow to communicate between two diagrams.

5.8.4 Scoping

The equation sets present in the same scope (or sub-scope) of the scope that introduced the diagram can refer to the flows defined or consumed within this diagram.

5.9 Default parameter value

When declaring the formal inputs of an operator declaration, it is possible to introduce a default expression that will be taken by the instances of this operator for which the input is not connected. The default expressions are limited to constant ones (i.e. an expression that could be used in the definition of a constant) which allows this expression to be defined in all the operator instance places. As the default is defined at the instanciation point of the operator, this construct also applies to imported operators, in this case the default value is an artefact that is part of the model only and that does not impact the definition in the target code.

As a first example, consider the function below, it takes a `point` in a three dimensional space and the three coordinates of a vector in this space and returns `new_point` which is the point reached when moving from `point` along the specified vector `(dx, dy, dz)`. The coordinates of the vector have all a default value which allows to not connect the dimensions that are not affected by the vector one want to apply.

```
group coord = (x:float32, y:float32, z:float32);

function move (point: coord;
               dx: float32 default = 0;
               dy: float32 default = 0;
               dz: float32 default = 0)
returns (new_point: coord)
  new_point = (x: point.(x) + dx,
               y: point.(y) + dy,
               z: point.(z) + dz);
```

given this definition, all the following instances are correct and semantically equivalent:

- `move ((p), 1.2, 2.5, 0)`
- `move ((p), 1.2, 2.5)`
- `move ((p), dy:2.5, dx:1.2)`
- `move (point: p, dy:2.5, dx:1.2)`

Here is another example with a node defining a counter with a configurable start value which default value is `0` and a configurable step with a default value of `1`:

```
node count (start: int32 default = 0; step: int32 default = 1)
  returns (o: int32)
  o = start -> (step + pre o);
```

Here are different instances of this node and a prefix of the sequences they define:

instance	defined sequence
`count(3,5)`	3, 8, 13,...
`count(5)`	5, 6, 7,...
`count(start:5)`	5, 6, 7,...
`count(step:2)`	0, 2, 4,...
`count()`	1, 2, 3,...

A default value can also be a group as in the following example:

```
group coord = (x:float32, y:float32, z:float32);
function move2 (point: coord; vect: coord default = (x:1,y:0,z:0))
returns (new_point: coord)
```

```
new_point = (x: point.(x) + vect.(x),
             y: point.(y) + vect.(y),
             z: point.(z) + vect.(z));
```

The function `move2` moves a point as `move` but in this case, the vector is specified as a unique parameter `vect` that is a group of type `coord`. A default value is specified for the translation vector is given. The difference with the previous definition is that it allows only two forms of instance with either all the parameters or the point only; the two instances below are semantically equivalent:

- `move2 ((p), (x:1, y:0, z:0))`
- `move2 ((p))`

5.10 Inline operators

Operators can be declared as expanded (or inlined) by preceding their declaration with keyword **inline**. This information is used by the causality analysis to decide whether the model is correct for that analysis.

For example let us consider the following operator that implements a custom unit delay on Boolean that is initialized to **false**:

```
inline node unitDelayF (a: bool) returns (o: bool) o = false pre a;
```

because it is declared as inlined, the causality analysis can consider that its output `o` does not depend instantaneously on its input `b`. This allows to use it as a **pre** to cut algebraic loops like in this equation:

```
...
o = a or unitDelayF(o);
...
```

Removing the keyword **inline** would just consider as incorrect a model with an equation like the one above.

6 Scade 6 changes

This section contains Scade 6 constructs that are modified or not present in Swan.

6.1 Remove fby

In Swan a unit delay is always obtained with **pre** either initialized or not. Thus as in Scade 6, `pre x` represents an uninitialized delay; the resulting flow can be given a first value thanks to operator `->`, for instance: `i -> pre x`. The **pre** operator in Swan admits an infix binary form `i pre x` that is a shortcut for `i -> pre x`. In its binary form, the operator is right associative which provides a lightweight syntax form for initialization sequences. For instance, the expression `3 pre 2 pre 1 pre x` represents the flow

3	2	1	x_0	x_1	$\cdots$	x_n	$\cdots$

A delay longer than one cycle that was expressed with a Scade 6 **fby** operator is written in Swan using a temporal window (long enough to contain the delay of interest) combined with a static projection. This separation allows to explicitly share in the model the window on a flow between different delay lengths.

6.2 Remove partial iterators

These primitives are now covered by **forward**, see 5.7.8.

6.3 Remove "sharp" operator

This operator is not standard in the literature of Boolean function, it corresponds to a particular use of the more classical *population count* function on bit vectors that computes the number of bits that are true.

In the past few years, the safety team assessed that this operator is not really understood by most of the SCADE users.

For these reason it is removed from the list of primitives and will be encoded in the core language and given as an example or in a library.

6.4 Remove predefined times operator

The **times** operator was treated as a primitive in SCADE 6 to give it an infix syntactic form and its definition is given in SCADE. We propose to provide it as a library operator and abandon the infix form. Thus `5` **`times`** `c` in SCADE 6 becomes `_times(5, c)` in SWAN. The library shall also provide a specialized version for which the condition is true.

6.5 Remove signal and keep emission for Boolean flows

The declaration of signals and the name "signal" disappears from the language. The possibility to define a Boolean flow by emitting it in several points of the design still exist but comes with some changes:

- SCADE 6 signals declaration and signal presence expression are removed;
- emission **emit** `'a` **`if`** `c` is correct if `c` is Boolean and `'a` is the name of a Boolean flow;
- a Boolean flow is defined either by equations (one active definition per cycle) or by emissions but cannot use both style.

The rationales for this change are the simplification of the language and the preparation for the integration of continuous aspects (aka. *hybrid language*) where it will be convenient to use the word signal to talk about continuous time functions and flows for discrete ones.

6.6 Syntactic changes in expressions

6.6.1 New syntax for transposition

The **transpose** operator takes two immediate integer values that identify the two dimensions to swap by their position in the multi-dimension array type (aka. *tensor* when items are numerical). In the new syntax, **transpose** takes none, one or two dimension positions. When the specification of the dimension are missing, the implicit values are:

- 1 and 2 in case nothing is specified;
- 1 for the unspecified dimension in the case where only one is given.

6.6.2 Change prefix form syntax

This point is now covered by the syntactic forms for *partial application*.

6.6.3 Remove `make` and `flatten`

The name **flatten** is not very appropriate for this operation and is better used for flattening 2 dimensional arrays on a single dimension. This operation is dangerous when applied to an array type since it returns a number of flows that is equal to the size of the type, which can lead to an explosion of the model size. The case of `make` for arrays is actually useless, array constructor does the job without having to give a type.

These two SCADE 6 construct have equivalent constructs for structures that can be seen as group to structure convertors.

6.6.4 Operators taking several groups

In SCADE 6, **fby** and **merge** operators take several groups as input and have a prefix form. A semicolon is used to separate the different kind/group of parameters: **fby**`(a; N; i)` and **merge**`(c; a; b)`.

In SWAN these cases are cover in the syntax by allowing to apply an operator instance in its prefix form to several groups. So **merge**`(c)(a)(b)` is now the way to pass these groups. **fby** is no more present in the language it is replaced by **window** that, as **fby** takes two groups and a size; it nows follows the general form for instance application **window**`<<N>>(I)(e)` as presented in 5.1.

Note: it is not yet possible to define an operator that takes several groups, so this form is only used for some primitives and is considered incorrect for user defined operators.

6.6.5 New syntax for numerical cast

Numerical cast in SCADE 6 was written `(e:T)` and is now written `(e:>T)`. The semantics is the same, the rationale for this change is the extension of the groups that can now have labels and for which it has been decided to use colon and parentheses. This new syntax solves the ambiguity introduced by the extension of groups.

6.7 Changes in control blocks

In SCADE 6 the control blocks (**activate** or **automaton**) have a return clause that allow to partially specify the computed flows; those not specified are deduced based on the equations that define them in the sub-scopes. In SWAN the list of defined variables is either absent or complete (no more `..`). In this particular form of equations, the right-hand-side is called *definition by case*. The idea is that activation and automata allow to provide different definitions for a variable that are exclusive and their validity depend on a case based analysis. The selection of a case can be based on the value of a flow (see example `solve2nd` below) or on a dedicated logic, possibly sequential, specified as an automaton[5]. The case analysis decides which scope contains the equations that hold for the current cycle, in this sense the corresponding scope is said to be *activated*.

Here is the example of the second degree equation resolution which is a typical example of case based definition. Only the activated scope is considered for the cycle, this prevents from applying `sqrt` function out of its definition domain.

```
function solve2nd(a: 'T; b: 'T; c: 'T)
  returns (xr: 'T; xi: 'T; yr: 'T; yi: 'T) where 'T float {
  var
    delta : 'T ;
    two_a : 'T ;
  assume $second_degree : a <> 0.0;
  let
```

[5]This form is sometimes called *mode automata*.

```
        delta = b*b - 4.0*a*c ;
        two_a = 2.0 * a;
        xr, xi, yr, yi: -- this line is optional, if present, consistency is checked
        activate $delta_cases
          if (delta > 0.0)
          then {
            var d : 'T;
            let
              d = sqrt(delta) ;
              xr, xi = ( (-b-d)/two_a , 0.0 ) ;
              yr, yi = ( (-b+d)/two_a , 0.0 ) ;
          }
          elsif (delta = 0.0)
          then {
            let
              xr, xi = ( -b/two_a , 0.0 ) ;
              yr, yi = ( xr, yr ) ;
          }
          else {
            let
              xr, xi = ( -b/two_a , sqrt(-delta)/two_a ) ;
              yr, yi = ( xr       , - xi ) ;
          };
    }

    function sqrt(x: 'T) returns (y: 'T) where 'T float;
```

6.8 Optional type in local declaration

The rule for *var_decl* has changed such that the type is now optional. Type can be omitted in local declarations (**var** sections); in this case, the type information is deduced from the definition. This inference assumes that a local identifier represents a unique flow and not a group. Explicit type declaration is still required for operators inputs and outputs; this requirement is verified as a post-syntactical check.

6.9 *n*-ary forms of binary operators

SWAN integrates in the syntax of expressions the *n*-ary extensions of some binary operators. An operator *bop* extends (syntactically) to an *n*-ary interpretation denoted (*bop*) that is interpreted as:

(*bop*)(e1, e2, e3, ···, en)

is syntactically interpreted as

(··· ((e1 *bop* e2) *bop* e3) *bop* ··· en)

This extension is defined for the following operators:

- addition (+)
- multiplication (*)
- array concatenation (@)

- **(and)**
- **(or)**
- **(xor)**
- `(land)`
- `(lor)`

This list is specified in the BNF by the definition of the non-terminal *n_ary_op*.

Open question: this list comes from Scade 6 and should be reconsidered; for instance it contains **xor** but not `lxor`. It is not clear that this extension is relevant for bitwise operations. The idea here is to reduce this list to the useful cases.

6.10 Changes in state machines

6.10.1 Remove final states and synchro transitions

In Scade 6 a state can be marked as `final` which allows to specify a `synchro` weak transition on the level above. This construct is not present anymore in Swan because it appears to be quite rare in practice and it is very easy to implement without dedicated support.

6.10.2 Additional syntax for transitions

It is possible to declare a transition in a state machine without attaching it to its source state. This can be done by giving an explicit priority (an integer literal) and the transitions with the same source are inspected in the increasing order of priority. Since it is neither allowed to mix strong and weaks nor detached and state attached transition form, it is always possible to order transitions as long as two transitions with the same source have different priorities.

```
automaton
  initial state S1:
  unless
     if (a) restart S2;
     if (b) resume S3;
     if (c) restart S4;
  let ...;

  state S2: ...
...;
```

is equivalent to

```
automaton
  initial state S1:
  let ...;

  :1:  S1 unless if (a) restart S2;
  :10: S1 unless if (c) restart S4;
  :5:  S1 unless if (b) resume S3;
```

```
    state S2: ...
  ...;
```

6.10.3 Transitions without guard

A transition without guard is allowed in Swan for simple transitions, i.e. without forks. In absence of guard, the transition is always fireable:

```
  ...
    unless
        ...
        restart S;
  ...
```

is equivalent to

```
  ...
    unless
        ...
        if true restart S;
  ...
```

6.10.4 Mix of weak and strong

In Swan it is not allowed to mix weak and strong transitions in an automaton. The justifications for this restriction are:

- Understandability and maintainability of the model, because only one transition can be fired per cycle, the mix gives surprising (still well defined) behaviours.
- Quality of the generated code because when mixing these two kind of transitions, the generated code cannot contain less than two "switch-case", one to inspect the strongs and one to compute the cycle and inspect the weaks.

Note that most of the time the mix can be easily avoided, the rule of thumb is to use weaks when the transition guards are based on what occured in the source state while the strongs are used to handle conditions that are not computed by the considered automaton. To deal with these two levels it is recommended to use state machine hierarchy with a top level automaton that handles external flows with strong transitions and, in its states, automata using weaks forms to manage the internaly computed decisions.

6.11 Change syntax for operator activation default

In Scade 6 the activation of an operator exists with two variants: one with default value when not activated and the second that holds the last output value and the provided value is used to give a first value to cover the case where it is not activated in the first cycle. The syntax for this second case has changed in Swan, it is now:

```
      (activate N every c last i)
```

where `i` is the expression that gives the initial value.

6.12 Changes on packages

Packages are replaced by modules and namespace as defined in 4. Unlike SCADE 6 packages a module cannot be opened in another module, the in SWAN the use of a module follows the principles sketched in 4.3. Another change compared to SCADE 6 is that two modules cannot refer circularly to each other (if a module A uses items of a module B then it is not allowed in B to refer items in A).

Note: In the present version, nesting modules is not possible while it was allowed for packages in SCADE 6.

6.13 Changes in types

6.13.1 Change structure type equivalence

In SCADE 6, two structure types are equivalent if they define the same list of fields i.e. same field names and with equivalent types in the corresponding fields. For instance, let us consider the piece of text below:

```
type
  Coordinate = {x : float32, y : float32};
  Vector = Coordinate;
  Position = {x : float32, y : float32};
```

with SCADE 6 these three types are equivalent (i.e. any flow declared as one on those types can be used where one of the two other one is required). In SWAN this is not the case anymore, `Vector` and `Coordinate` are equivalent since one just aliases the other, but `Position` type has no other equivalent type here.

As a consequence, to construct a new structured value in SWAN, it is necessary to choose the targeted type. For instance the following SWAN function `F` is well typed:

```
function F (a: float32; b: float32; c: bool) returns (o: Vector)
  o = if c then {a, b}:Position else {x:0.0, y:0.0}:Coordinate;
```

In SCADE 6, the expression `{a, b}:Position` (resp. `{x:0.0, y:0.0}:Coordinate`) was written `{x:a, y:b}` (resp. `{x:0.0, y:0.0}`) which is not even syntactically correct in SWAN. And it was allowed to declare a variable with a structure type without first defining any structure type first (`v : x :` **float32**`, y :` **float32**`;`). The rationale behind this change is threefold:

- in many use cases of an anonymous structure type it is possible to replace it by a group,
- naming structure types is consistent with union types and
- naming structure types prepares separate compilation as it reduces the number of arbitrary names generated by the code generator.

6.13.2 Remove numeric imported types

Numeric imported types were introduced in the early versions of SCADE 6 when the language had only two primitive numeric types (**int** and **real**) to facilitate the use of different numeric types in a single model. The latest release of SCADE 6 introduces a family of integer types of different sizes and sign, and two floating types (**float32** and **float64**). SWAN continue with these sized numerical types as primitive, so there is no need to continue supporting numeric imported types.

6.14 Imported code and type

In SCADE 6, imported elements are marked with keyword **imported**, it is then checked that they do not have definition. In SWAN a module defined by an interface only is considered imported and an operator, a type or a constant declared without definition is also considered imported. In SWAN, the keyword **imported** is not used anymore.

6.15 Multi-accumulators handled as groups

example:

```
group G = (min: int8, max: int8, sum: int32);

function minMaxSum (accIn: G; x: int8) returns (accOut: G) {
  var m; M; s;
  let
    m, M, s = accIn.(min, max, sum);
    accOut = (min: if x < m then x else m,
              max: if x > M then x else M,
              sum: s + (x :> int32));
}

function root (A: int8^100) returns (o: G)
  o = (fold minMaxSum)<<100>>((min: 127, max: -128, sum: 0), A);
```

The example `fast_exp` introduced in 5.7.5 can be reformulate with multi-accumulators:

```
function fast_exp (x: 'T; p: uint8) returns (x_pow_n: 'T) where 'T numeric
  x_pow_n =
    ((fold
      (function (a:(u: uint8, v: 'T, w: 'T))
       returns  (b:(u: uint8, v: 'T, w: 'T)) {
         let
           b = (u  : a.(u) / 2,
                v  : a.(v) * a.(v),
                w  : if a.(u) mod 2 = 1 then a.(w) * a.(v) else a.(w));
       })
    ) <<8>>((u: p, v: x, w: 1))).(w);
```

6.16 Causality and inlining

In SCADE 6, the causality (thus the correction) of a model was expressed in the language specification regardless of any implementation constraint i.e. supposing all the operator instances as inlined. Then the code generator allows to specify operators to be expanded and other not which could lead to an incorrect model since requiring to translate an operator into one C function introduces causality constraints for the code generator. In some cases it leads to models that are accepted or not for causality reasons depending on the command line.

In SWAN, we propose to explicitly specify the operators that have to be considered as expanded by the causality analysis. The rationale behind this change is that there may be two reasons for expanding an operator:

1. because it is necessary to break an algebraic loop or
2. for better performances (in a trade-off code size / execution time).

With this new feature, the first is now expressed in the model and the second is just a code generator feature.

6.17 Change in constant declaration

In SWAN it is allowed to define a constant with an expression only as long as a monomorphic type can be infered from the expression. For example, the following constant definitions are correct:

```
type
  cart   = {x: float32, y: float32};
  coord2D =
      Polar     {r: float32, t: float32}
    | Cartesian {cart}
    ;

const
  b = true;
  N = 23_ui8;
  Pi = 3.1415_f32;
  P = {x: 42, y:7.6}:cart;
  Origin = Polar{r:0, t:0};
```

while the following ones are incorrect:

```
const
  N = 23;
  Pi = 3.1415;
```

because the expressions that define those constants are polymorphic.

6.18 Scopes in operator expressions

An operator instance is made of an operator (*operator*) which itself contains one operator expression (*op_expr*). The operator expression allows to stack higher-order forms (iterators, activate, restart), anonymous operator and partial application over a prefix operator. Among these cases, the activate, the restart and the partial application contain flow expressions; all these expressions are considered as beeing in the same scope. This means that these expressions are on the same clock and are not sensitive to an activate or restart appearing above in the operator expression they belong to. For instance:

```
y = (activate
      (restart Op every count() mod 10 = 0)
     every count() mod 2 = 0) (...);
```

where `count()` is an incrementing counter, is equivalent to the following equations:

```
r = count() mod 10 = 0;
a = count() mod 2 = 0;
y = (activate
      (restart Op every r)
     every count() a) (...);
```

An example with a partial application:

```
y = (restart (Op\(_,0->x) every a))(...);
```

behaves the same as:

```
xx = 0 -> x;
y = (restart (Op\(_,xx) every a))(...);
```

The case of an anonymous operator is different; anonymous operators introduce a scope (as any operator) and the expressions it contains cannot, in general, be extracted without affecting the behaviour.

6.19 Pattern list in a case expression

It is not allowed anymore to have a **case** expression with a unique default branch, i.e. the following expression is incorrect in Swan while it was correct in Scade 6:

```
(case x of | default: e)
```

that can also be written:

```
(case x of | _: e)
```

The rationale behind this choice is to favor type inference rather the ability to write these degenerated forms of selections. Indeed, if `x` is only used in this expression and has no declared type, the inference should conclude that it is of type `integer`, **char** or some enumeration (or union). Since this is the only case where such a complex constraint is needed and since it does not introduce any interesting expressivity, it was decided to not accept this degenerated case.

6.20 New kind of assertion

In Scade 6 two kind of assertions are allowed: **assume** and **guarantee**. Swan introduces another kind, called **assert** with the same form as the other two, the kind expresses an intend that may introduce different support and features in verification tools.

One may wonder why these constructs are part of the language since they do not affect the input/output behaviour of an operator. The reason for that are:

- expressions in the assertions are part of the language and their correctness must be checked with the same criteria as for any other expression;
- as a modeling language, Swan is not dedicated to code generation only it is also the object of verification activities that may need some support in the language; assertions fall in this case.

Let us consider an operator `Op` and a snippet of its definition:

```
node Op (i1: float32; i2:T2, ...) returns (o1: uint32; o2: U2, ...) {

  assume Positive: i1 >= 0.0;
  guarantee Even: o1 mod 2 = 0;

  var x; inv_x;
  let
    x = i1 + 0.1;
    inv_x = 1/x;
  assert NotNull: x <> 0;

  o1 = (inv_x : uint32) land 0xfffffffe;
```

```
        ...
    }
```

The intends expressed by the different kinds of assertions when used in the body of `Op` are:

- **assume**: refines the input domain of `Op` defined by the input declared types;
- **guarantee**: defines the invariants on the outputs that are expected to hold for `Op` i.e. that are part of its functional requirements.
- **assert**: defines an invariant of the implementation that is not part of `Op` functional requirement.

This separation of the contract and the internal invariants provides a clear distinction between what an operator is supposed to do (**assume**/**guarantee**) and the other properties of its implementation.

6.21 Change initialization analysis of divisions

In SWAN the second operand of the binary operators '`/`' and '**mod**' must be a flow with a defined first value. Using the notation of the initialization analysis, this means that these two operators have the following type:

$$\forall \delta . \delta \times \mathbf{0} \longrightarrow \delta$$

With this restriction it is not possible to divide by the output of an uninitialized unit delay; `1/(`**pre** `e)` is now rejected by the initialization analysis.

This change is made because these two operators are partial i.e. not mathematically defined when the second argument is zero. Allowing an uninitialised flow in this place lead, in practice, to have an arbitrary value as a divisor thus possibly a zero. This restriction is consistent with any kind of verification of absence of division by zero based on the model only (using a model checker for instance).

end of document

DOCUMENT RELEASE FORM

Repository:	repository
Title:	The Swan Language Primer
(version 2025.0)	
Identifier:	
Version:	2.1
Date:	2024-10-23
Classification:	NDA RESTRICTED
File name:	
Total pages: *(Including this one)*	74

Abstract

This document is an excerpt of the Swan language document reduced to:

- the syntax;
- illustration of language features that are new compared to Scade 6;
- the changes compared to Scade 6.

Document Approval

Author(s)	Jean-Louis Colaço
	Olivier Andrieu
	Loïc Wagner
	Bruno Martin
	François Carcenac
Language Approval	LANGUAGE EXPERT
	Jean-Louis Colaço

Distribution List

Project team