

# THE PROGRAMMING LANGUAGE OPAL

5<sup>th</sup> corrected edition

The OPAL Group  
Fachbereich Informatik  
Technische Universität Berlin

Edited by PETER PEPPER

November 1997

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Structure Signature</b>	<b>5</b>
2.1	Signature . . . . .	5
2.1.1	Sorts . . . . .	6
2.1.2	Operations . . . . .	6
2.2	Free Types . . . . .	6
2.3	Export and Import . . . . .	7
<b>3</b>	<b>Structure Implementation</b>	<b>9</b>
3.1	Signature Extension . . . . .	10
3.2	Data Types . . . . .	10
3.2.1	The Definition of Data Types . . . . .	10
3.2.2	Free Types and Data Types . . . . .	11
3.3	Operations . . . . .	12
3.3.1	Standard Definitions . . . . .	12
3.3.2	Pattern-Based Definitions . . . . .	14
3.4	Expressions . . . . .	15
3.4.1	Atomic Expressions . . . . .	15
3.4.2	Tuples . . . . .	16
3.4.3	Function Applications and Mixfix Notation . . . . .	16
3.4.4	Lambda Abstractions . . . . .	18
3.4.5	Sections . . . . .	18
3.4.6	Case Distinctions . . . . .	19
3.4.7	Extended Expressions . . . . .	21
<b>4</b>	<b>Parameterization</b>	<b>22</b>
4.1	Parameterized Structures . . . . .	22
4.2	Instantiation . . . . .	23
<b>5</b>	<b>Names and Scopes</b>	<b>25</b>
5.1	Names . . . . .	25
5.1.1	Origin . . . . .	25
5.1.2	Kind . . . . .	26
5.1.3	Instantiation . . . . .	26
5.1.4	Overload Resolution . . . . .	27

5.2	Scopes . . . . .	27
5.2.1	Global Names in a Structure . . . . .	27
5.2.2	Local Names . . . . .	28
<b>6</b>	<b>Programming in OPAL</b>	<b>30</b>
6.1	What is an OPAL Program? . . . . .	30
6.2	Input/Output . . . . .	30
6.2.1	A Simple Program . . . . .	31
6.2.2	State-Preserving Beyond Input/Output Boundaries . . . . .	32
6.2.3	Not Inherently Sequential Input/Output . . . . .	32
6.3	The Library . . . . .	33
<b>A</b>	<b>OPAL Syntax</b>	<b>34</b>
A.1	Definitions and general context conditions . . . . .	34
A.2	Structures . . . . .	37
A.3	Signature . . . . .	38
A.4	Free Types and Data Definitions . . . . .	39
A.5	Function Definitions . . . . .	41
A.6	Expressions . . . . .	43
A.6.1	Bracketing of Infix-Expressions . . . . .	46
A.7	Algebraic Properties . . . . .	47
A.7.1	Laws . . . . .	47
A.7.2	Formulas . . . . .	47
A.8	Partial Names . . . . .	48
A.9	Lexical Rules . . . . .	49
<b>B</b>	<b>Changes</b>	<b>52</b>
<b>C</b>	<b>Acknowledgement</b>	<b>53</b>

# Chapter 1

## Introduction

The OPAL project is an experiment that sets out to explore the optimal compilation of purely applicative programming languages. Hence, the language OPAL comprises those features of applicative languages that we consider essential for our research on optimal compilation. If our intention had been to design a language for actual software production, some design decisions would have turned out differently. We hope to produce (at some future date) a follow-up version of the language, including all the nice syntactic features that make a language truly comfortable to work with. It should not come as a surprise that OPAL resembles other applicative languages such as ML or — even more closely — HOPE.

In order to understand our main design choices more clearly, one should bear in mind the context into which we would like to embed OPAL. We envisage a programming environment in which program development starts from high-level algebraic specifications. These specifications are then transformed and refined in a stepwise process until they are brought into a constructive form. This is where OPAL should take over.

This methodological background suggests that a language like OPAL should comprise three components: a signature part, a specification part, and an implementation part. The specification aspects, in particular the compatibility between specification and implementation, will be the subject of a research project that we intend to conduct (in cooperation with others) in the near future. *The version of OPAL, as presented in this report, deals only with the constructive aspects of the language, i.e., the signature part and the implementation part.*

When using more traditional development methods, the programmer has to produce classical code from a specification, employing either transformation or verification techniques, such that the result runs efficiently and meets the specification. Our hope is that, for a certain class of constructive specifications, this task can be performed automatically by a compiler.

It was these considerations that motivated the following design choices:

- The overall appearance of OPAL programs is *strongly algebraic*. In particular, we have signatures and recursive equations.
- *Modularization* is an important element in modern specification and programming languages. Hence, OPAL programs are built up from structures (each consisting

of a signature and an implementation part) that are organized in a hierarchical usage relation, providing import and export interfaces.

- Applicative languages should treat functions as “first-class citizens”. Hence, OPAL provides *higher-order functions*, i.e., functions with functions as arguments and/or results.
- The orientation towards algebraic specification languages as well as the interest in efficient implementation leads to a *strong typing discipline*. This does not, however, rule out attractive features like parameterization and overloading.
- The need to produce truly efficient code does — as far as we are aware — make eager evaluation mandatory. In other words, we employ a *call-by-value* mechanism.
- Input and output are realized by means of referentially transparent input/output *continuations*.

## Overview

An OPAL program consists of a collection of *structures*. A structure is the OPAL counterpart of what is usually referred to as “module”, “cluster”, “package”, “encapsulation”, “class”, etc. Semantically speaking, structures are algebras. The structures of an OPAL program are in an acyclic hierarchical dependency relation, the “uses” or “import” relation.

An OPAL structure consists of a visible *signature part* and a hidden *implementation part*. These are discussed in sections 2 and 3.

Structures can be parameterized by sorts and operations. Parameterized structures are introduced in section 4.

Overloading and parameterization call for flexible naming of OPAL objects. These naming facilities are explained in section 5.

A description of how to construct an OPAL program rounds off this report.

A formal syntax, an availability notice, acknowledgements and a bibliography are given in the appendices.

## Remarks on the Syntax

When reading this report, the following syntactic conventions should be kept in mind.

OPAL distinguishes four kinds of lexical symbols:

- alphanumeric symbols such as: `hallo x3 1`
- graphic symbols such as: `++ % -- ==>`
- separators: *blanks* , ( [ ) ] ’
- strings such as: `"Hello World!"`

Two alphanumeric, two graphic symbols, or two strings have to be separated from each other by suitable separator symbols; however, the direct juxtapositioning of different kinds of symbols is allowed. Moreover two different symbols can be combined by an underscore.

*Examples:* The following compositions are legal without separators:

```
x++y      x-1      nat**bool
```

A composition of different symbols by an underscore which are interpreted as one lexical symbol looks like :

```
no_%      -_1      add_one
```

The following composition requires a separator:

```
<| :text**text->bool
```

□

Note that the *question mark* plays a special role: it may be appended directly to both alphanumeric and graphic symbols.

OPAL makes substantial use of keywords. *Keywords* are written in CAPITAL letters. They are reserved words.

There are two ways of writing *comments*. The first supports readability of layouts in print or on terminals; this kind is started by the symbol `--` (i.e., a double dash) and it is terminated at the end of the line. The second is more structure-oriented; it starts with the symbol `/*` and terminates with the symbol `*/`. This allows, in particular, “nested comments”. The comment symbols are keywords!

OPAL includes *denotations* which are given as strings, i.e., character sequences enclosed in double quotes, e.g., `"Hello World!"`.

OPAL *identifiers* are arbitrary graphic or alphanumeric character sequences (except for the reserved words); for example, `2`, `+` and `OPAL` are identifiers.

The items in a *tuple* (e.g., parameters of a function) are separated by commas and, if necessary, enclosed in parentheses. By contrast, mere *collections* of items whose order is irrelevant are enumerated without commas. Tuples are used only where necessary, e.g., for function parameters; collections, on the other hand, are used wherever possible.

## Chapter 2

# Structure Signature

The major programming units of OPAL are *structures*. They define data elements as well as operations. The *interface* of a structure, i.e., the components that are visible to the outside world, is given by the *structure signature* (which plays a similar role to the “definition modules” in MODULA2). It consists of the following items:

- the *signature* describes the “syntactic aspects”, i.e., the sorts and the operation (together with their functionalities);
- the *free types* give a certain structural appearance to the data elements.

*Example:* The following structure **Text** is intended to give an initial intuitive introduction to the OPAL syntax. It introduces sequences of characters and operations on them.

```
SIGNATURE Text
IMPORT Char ONLY char           -- imported
      Nat  ONLY nat            -- signature
-- free type
TYPE text == empty             -- empty text
      ::(first: char, rest: text) -- appending a character
-- additional operations
FUN ++      : text ** text -> text -- concatenation
      revert : text          -> text -- revert
      length : text          -> nat  -- length
      =      : text ** text -> bool -- equality
```

□

## 2.1 Signature

The two components of a signature are:

- a set  $S$  of *sorts*, where a sort is a name for a set of values;
- a set  $O$  of *operations*, where an operation is a name for a constant or a function. Each operation  $o \in O$  has a *functionality*.

The sorts and operations of a structure come from two sources: either they are listed explicitly in the signature (including free types), or they are inherited from the imported structures.

The scope of the sorts and operations is the whole structure.

OPAL is a strongly typed language which does, however, allow overloading and parameterization. The basic requirement, therefore, is that each name can be unequivocally identified. The detailed conditions that follow from this principle are listed in section 5.

### 2.1.1 Sorts

*Sorts* are simply names. Semantically, they denote carrier sets. These carrier sets may consist of elementary values such as the natural or real numbers, or they may consist of complex data structures such as sequences, trees, graphs, or arrays.

*Examples:* The sorts of the booleans, the natural numbers, and texts are denoted by

```
SORT bool nat text
```

□

### 2.1.2 Operations

*Operations* are names for elementary values or functions. Elementary values are elements of carrier sets; functions are mappings from arguments to results. Each operation has a functionality, which describes its sort or its domain and range.

Functions are “first-class citizens” in OPAL. Hence, functionalities exhibit a deeper structure than the simple “argument-tuple-plus-result” pattern: we have Cartesian products and mappings, built on top of sorts.

*Examples:*

```
FUN 0 1 2 3 4 5 6 7 8 9:nat      -- constants
FUN pi:real                      -- constant
FUN divmod:nat**nat->nat**nat    -- function
FUN filter:(char->bool)->(text->text) -- higher-order function
```

□

Cartesian products may not be nested, e.g. products such as  $A^{**}(B^{**}C)^{**}D$  are not permitted. The mapping operator  $\rightarrow$  associates to the right; i.e.,  $A \rightarrow B \rightarrow C$  is equivalent to  $A \rightarrow (B \rightarrow C)$ . Finally, the empty tuple is only allowed as the domain of a function, thus characterizing it as a nullary function.

## 2.2 Free Types

Recursive type definitions are an elegant means of defining many kinds of data structures. Moreover, they support clear function definitions by means of structural induction (“call-by-pattern”; see section 3.3). However, the principle of information hiding requires that the internal structure of data elements should not transit the interface. OPAL therefore



provides the concept of *free types*, allowing a data type to be viewed as if defined by a recursive type definition, irrespective of the actual implementation which has to be *behaviourally equivalent* (see section 3.2).

Only one free type may be given for each sort. Each free type induces its signature (see below).

*Example:* The free type `text` is recursively defined with `empty` and `::` as the free constructor operations, and `empty?` and `::?` as its discriminator operations. The selector operations are `first` and `rest`.

```
TYPE text == empty
          :: (first:char, rest:text)
```

The type `text` automatically induces the following signature:

```

SORT text                                -- texts
FUN empty  : text                        -- the empty text
  ::       : char**text->text           -- prefixing a character
empty?    : text->bool                   -- test for emptiness
::?       : text->bool                   -- test for nonemptiness
first     : text->char                   -- the first character
rest      : text->text                   -- the remaining characters
```

The operations (have to) obey the following laws:

$$\begin{aligned}
\forall e, s & : \text{empty?}(\text{empty}) \wedge \neg \text{empty?}(e :: s) \\
\forall e, s & : ::?(e :: s) \wedge \neg ::?(\text{empty}) \\
\forall e, s & : \text{first}(e :: s) \equiv e \\
\forall e, s & : \text{rest}(e :: s) \equiv s \\
\forall s & : ::?(s) \Rightarrow \text{first}(s) :: \text{rest}(s) \equiv s \\
\forall s & : \text{empty?}(s) \vee ::?(s)
\end{aligned}$$

□

Free types are a counterpart of data types; see also section 3.2.

## 2.3 Export and Import

The *export* of a structure is its complete signature part. Exported sorts and operations of a structure can be *imported* by another structure.

The import becomes part of the signature. Hence, all sorts and operations that are imported in the signature part are automatically re-exported. This has the effect that it is possible, say, to import from the structure `Text` the sort `nat` from `Nat`.

The basic syntactical form of an import from a structure is the *selected import*: only the explicitly listed sorts and operations are imported. When an overloaded identifier is listed, all its (matching) instances are imported (see section 5.1.4). The alternative form of an import is the *complete import* which is only allowed in the implementation part of a structure.

*Example:* In the structure **Text**, we import from **Nat** only the sort **nat**.

```
IMPORT Nat ONLY nat
```

However, if we wish to import everything from **Nat** in the implementation part, we can write

```
IMPORT Nat COMPLETELY
```

□

The same structure may be imported several times. If the selections are not disjoint, the common sorts and operations are actually imported only once.

Every structure implicitly imports all sorts and operations from the predefined structures **BOOL** and **DENOTATION**.

## Chapter 3

# Structure Implementation

Implementations provide the actual definitions of the various items of a structure. The two components of an implementation are:

- a set of *carrier sets*, which are defined by means of data definitions;
- a set of *values* and *functions*, which are defined by means of (recursive) equations.

These carrier sets, values and functions must be in a one-to-one relationship with the sorts and operations listed in the signature of the structure.

Continuing our example, the implementation part of the structure `Text` may be defined as follows:

```
IMPLEMENTATION Text
IMPORT Char ONLY =                                -- additionally needed
      Nat COMPLETELY                             -- signature
-- data definition
DATA text == empty
      ::(first:char, rest:text)
-- concatenation
DEF t1++t2 == IF t1 empty? THEN t2                -- simple definition
      ELSE first(t1)::(rest(t1)++t2) FI
-- length
DEF length(empty) == 0                            -- pattern-based
DEF length(_ ::t) == succ(length(t))              -- definition
-- revert
DEF revert(t) == rev(t,empty)
FUN rev:text**text->text
DEF rev(empty,z) == z
DEF rev(c::t,z) == rev(t,c::z)
-- equality
DEF empty==empty == true
DEF (_ :: _)=empty == false
DEF empty=(_ :: _) == false
DEF (c1::t1)=(c2::t2) == IF c1=c2 THEN t1=t2 ELSE false FI
```

### 3.1 Signature Extension

The implementation part inherits the signature (including the induced signature of the free types) of the corresponding signature part, but it can be enriched by hidden signatures and free types which may also come from additional imports.

*Example:* The implementation part of the structure **Text** has the additional import of **Nat** and **Char**, and the hidden function **rev**. □

### 3.2 Data Types

The types (here: free types and data types) of OPAL correspond to the classical mathematical constructions of direct product and direct sum. As a result, we obtain the canonical operations such as projection functions (in lieu of special language features), etc. This is of particular benefit in connection with the direct sum, where we have a very clear and precise conception.

The syntax of type definitions is chosen in such a way that it comprises all the necessary information in a very compact way. The signature of these operations is automatically derived from the type definition.

#### 3.2.1 The Definition of Data Types

A *data definition* introduces a *sort* (as its left-hand side) together with a structural description of the corresponding data elements (as its right-hand side):

- A data definition consists of the direct sum of one or more variants.
- Each *variant* is either a constant or a direct product of one or more components.
- Each *product* consists of the description of a constructor operation and a selector operation for every component.
- For each variant, we have a *discriminator* operation that tests whether a given data element belongs to the variant or not. The identifier for this discriminator operation is obtained by simply appending a question mark to the corresponding constructor.

The variant names, the constructor operation as well as the selector operations and the sorts of the components. The signature of these operations is automatically deduced from the data definition.

*Examples:* The data type **text** is recursively defined as a sum: **empty** and **::** are the constructor operations, and **empty?** and **::?** its discriminator operations. The selector operations are **first** and **rest**.

```
DATA text == empty
           :: (first:char, rest:text)
```

The signature of the induced canonical operations is like the one for the equivalent free type (see section 2.2).

The following (enumeration) type simply introduces two constants:

```
DATA bool == true false
```

A data type `player` is a typical instance of a product type. Note that the sort and the constructor operations may have identical identifiers (but this need not be so). Note also that functions are admitted as components of products.

```
DATA player == player(name:text,
                      age:nat,
                      statistics:game->points)
```

The same selector identifier may appear in different variants of a sum. It is then actually one selector:

```
DATA address == short(name:text, prename:text, street:text)
               long( name:text, prename:text, street:text,
                   state:text, tel:nat)
```

Note that the same constructor name may not be used for short and long, because the resulting discrimination operations would have the same name (see section 5.1.4).  $\square$

Except for the canonical operations, no additional operations are automatically defined on types: neither an equality nor an order or anything else. The main reason for such a rigorous decision is that OPAL types can incorporate functions, and there is no computable equality on functions.

### 3.2.2 Free Types and Data Types

A data definition looks very similar to a free type. Both concepts are strongly connected, but there are also essential differences. Both a data definition and a free type automatically induce the signature, consisting of the appropriate sort and the canonical constructor, selector and discriminator operations. However, a data definition also constitutes a concrete implementation for the data elements and for the operations. This is not the case for free types. It is therefore the programmer's task to explicitly provide definitions of the sorts and the canonical operations of the free types.

Free types must be implemented by behaviourally equivalent data types (see [15]). It should not be possible to distinguish between the free type and the data type by evaluating exported functions.

*Example:* Consider an alternative data definition of texts, where texts are implemented by a sequence of bounded character arrays. For details of (parameterized) sequences and arrays, see section 4.

```
DATA text == text (chunks:seq[chunk])
DATA chunk == chunk(chars:array[char], from:nat, to:nat)
```

The operations of the free type now have to be programmed on the basis of the given data type; we can do this using the above type structure:

```

DEF empty == text(<>)
DEF c::t == IF chunks(t) <>?      THEN text(singleton:: <>)
              ELSE IF from(fst) = 0 THEN text(singleton::chunks(t))
              ELSE text(new::rest(chunks(t)))
              FI WHERE fst == first(chunks(t))
              new == chunk((chars(fst),from(fst)-1):=c,
                          from(fst)-1,to(fst))
              FI WHERE singleton == chunk(init(10,c),9,9)
-- etc.

```

From the user's point of view, only the free type as given in section 2.2 is visible. Hence, the free type and the data type are indistinguishable. But note that, for the above implementation, the laws of the free type that state equivalences on texts only hold for behavioural equivalence.

The equivalences

$$\begin{aligned}
\forall e, s & : \text{rest}(e :: s) \equiv s \\
\forall s & : ::?(s) \Rightarrow \text{first}(s) :: \text{rest}(s) \equiv s
\end{aligned}$$

actually stand for the behavioural equalities

$$\begin{aligned}
\forall e, s & : \text{first}(\text{rest}(e :: s)) \equiv \text{first}(s) \wedge \text{length}(\text{rest}(e :: s)) \equiv \text{length}(s) \wedge \\
& \text{empty?}(\text{rest}(e :: s)) \equiv \text{empty?}(s) \wedge ::?(\text{rest}(e :: s)) \equiv ::?(s) \\
\forall s & : ::?(s) \Rightarrow \text{first}(\text{first}(s) :: \text{rest}(s)) \equiv \text{first}(s) \wedge \\
& \text{length}(\text{first}(s) :: \text{rest}(s)) \equiv \text{length}(s) \wedge \\
& \text{empty?}(\text{first}(s) :: \text{rest}(s)) \equiv \text{empty?}(s) \wedge \\
& ::?(\text{first}(s) :: \text{rest}(s)) \equiv ::?(s)
\end{aligned}$$

□

Only free types allow function definition by pattern matching. If there is no free type defined, then the data type automatically induces a corresponding free type.

### 3.3 Operations

In OPAL, the notion of *operations* includes *functions* as well as *elementary values* (“constants”). Owing to the algebraic orientation of the OPAL programming style, function definitions are not written as lambda abstractions, but rather in the form of special equations. For these equations, there are two notational variants: *standard definitions* and *pattern-based definitions*.

#### 3.3.1 Standard Definitions

The standard form of a function definition has the following appearance:

$$f(x_1, \dots, x_m) \dots (z_1, \dots, z_n) == expression$$

That is, the left-hand side is the application of a function identifier to zero or more tuples of *parameters*, and the right-hand side is an arbitrary expression. (Function applications are described in section 3.4.3.) The functionalities of the parameters are induced by the functionality of *f* as listed in the signature. Empty parameter tuples, denoted by *()*, are admissible.

*Examples:* The following examples implement constants based on suitably defined functions.

```
DEF 1 == succ(0)
DEF pi == computePi()
```

The concatenation function on texts, with *++* as its identifier and *t1*, *t2* as parameters, is defined as follows:

```
DEF t1 ++ t2 == IF empty?(t1) THEN t2
                ELSE first(t1) :: (rest(t1)++t2) FI
```

A typical higher-order function is given in the following example: the function **filter** extracts from a text those elements that fulfil a certain predicate.

```
FUN filter: (char->bool)->text->text
DEF filter(p)(t) ==
  IF t empty? THEN empty
  OTHERWISE
  IF p(first(t)) THEN first(t)::filter(p)(rest(t))
                    ELSE filter(p)(rest(t)) FI
```

Since higher-order functions are allowed, it may — under certain circumstances — be possible to omit some of the final parameter tuples.

*Example:* Given the function **filter** (see above) and a predicate **isCaps** that tests whether a character is a capital letter, we can define a function that extracts all capital letters from a given text. In the signature, we have to write

```
FUN Caps: text->text
```

and the definition can be written equivalently in either of the forms

```
DEF Caps(T) == filter(isCaps)(T)
DEF Caps == filter(isCaps)
```

The scope of the parameters is the function definition. Therefore, the expression that forms this function body can be built up from the global names of the structure under consideration (i.e. import plus signature) together with the parameters.

### 3.3.2 Pattern-Based Definitions

Functions are often defined inductively on the structure of the underlying types. This paradigm is supported in OPAL by pattern-based definitions, which, essentially, look like normal definitions, with the exception that the left-hand side may also contain patterns in addition to parameters. A *pattern* is a constructor operation from a free type applied to arguments, which are either *pattern variables* or, again, patterns (nested patterns are allowed). All of the variables in a pattern must be distinct (because there is no predefined equivalence on sorts). Their scope is the equation. A special pattern variable *wildcard* (`_`) is a placeholder for pattern variables which are unreferenced at the right hand side of the definition.

*Example:* The function `filter` can be alternatively defined by patterns:

```
DEF filter(_)(empty) == empty
DEF filter(p)(x::t) == IF p(x) THEN x::filter(p)(t)
                        ELSE filter(p)(t) FI
```

Here, `empty` and `x::t` are patterns, and `x` and `t` are pattern variables. □

The set of equations need not be exhaustive; there may be values that do not match any of the given patterns. In this case, the function is simply undefined for such arguments (see also section 3.4.6).

The case analysis is realized by reference to a best-fit criterion: if a pattern is an instance of another pattern, the more specific pattern is checked first; otherwise, even if patterns overlap, the order in which they are checked is left open (see also section 3.4.6).

Using the abstract properties of free types, pattern matching is nothing more than a shorthand notation for an expression with explicit test and selector applications.

*Example:* The equality function (see introductory example to section 3)

```
DEF empty=empty == true
DEF (_ :: _)=empty == false
DEF empty=(_ :: _) == false
DEF (c1::t1)=(c2::t2) == IF c1=c2 THEN t1=t2 ELSE false FI
```

is translated into the following code (according to the given strategy):

```
DEF s1=s2 ==
  IF s1 empty? THEN IF s2 empty? THEN true
                    IF s2 ::? THEN false FI
  IF s1 ::? THEN IF s2 empty? THEN false
                IF s2 ::? THEN
                  LET (c1,t1) == (first(s1),rest(s1))
                    (c2,t2) == (first(s2),rest(s2))
                  IN IF c1=c2 THEN t1=t2 ELSE false FI
  FI
FI
```



□

References to the complete argument in the equation's right-hand side are facilitated by giving it a name in addition to the pattern.

*Example:* The function `fac` may be written as follows:

```
DEF fac(0)                == 1
DEF fac(m AS succ(n)) == m * fac(n)
```

□

*Note:* If we define a pattern as a pattern variable or a constructor operation from a free type applied to patterns, we can subsume the standard form of function definition as a trivial case of pattern-based definition. This is done from now on. Note that all parameters on the left-hand side are then viewed as pattern variables!

## 3.4 Expressions

*Expressions* are used to define the right-hand sides of function definitions. In general, they are evaluated each time the defined operation is applied. There is one exception, however: *constant expressions*, i.e., expressions that contain no free variables are evaluated during program initialization.

There are essentially six different forms of expressions:

- *atomic expressions*, i.e., applied occurrences of names or denotations,
- *tuples* of expressions,
- *function applications*,
- *lambda abstractions*,
- *case distinctions*, i.e., collections of guarded expressions and conditionals, and
- *extended expressions*, i.e., expressions with auxiliary names introduced in **LET** or **WHERE** clauses.

All expressions in OPAL are strongly typed.

### 3.4.1 Atomic Expressions

An *atomic expression* is either an applied occurrence of a name, or a denotation. In the former case, the sort of the atom is the sort of the name; in the latter case, it is the basic sort **denotation**.

*Examples:* Applied occurrences of names are

```
pi      1      +      x
```

Denotations are

```
"Hello World!"    "3.14159"    "2137"    "\t foo \n bar"
```

□

It should be noted that OPAL does not possess predefined number systems and the like. For example, the only way to represent numbers in the familiar decimal notation is to write them as denotations (and to apply an explicitly programmed conversion function).

### 3.4.2 Tuples

A *tuple* of expressions, enclosed in parantheses and separated by commas, is itself an expression. An empty tuple () is possible as the argument of a function application. The functionality of a tuple is the Cartesian product of the functionalities of its component expressions.

*Example:* The following expression consists of an argument value, a function and the corresponding result value:

```
(pi, sin, sin(pi))
```

Hence, its functionality is `real**(real->real)**real`.

□

*Note:* Tuples may be nested, but this is only a syntactic feature. The functionality of a tuple is always that of the unnested version.

### 3.4.3 Function Applications and Mixfix Notation

The *application* of a function to an argument expression is itself an expression. The resulting expression may again be a function and, thus, be applied in turn to an argument. This way, we obtain iterated function applications. The arguments of a function application must always be enclosed in parentheses. Function application associates to the left; that is  $f(x)(y)$  is the same as  $(f(x))(y)$ .

The semantics of the function application is *call-by-value*. In other words, both the function expression and the argument expression are evaluated before the actual application takes place. The functionality of a function application is the result functionality of the function as given in the signature.

To improve readability of OPAL programs, it is possible to write function applications in infix or postfix notation. The function symbol may be inserted at any point in the first argument list, i.e., if a function @ has the functionality:

```
FUN @ : A ** B ** C -> ...
```

then all the following expressions denote the same function application:

```
@(a,b,c)  (a)@(b,c)  (a,b)@(c)  (a,b,c)@
```

If the argument list to the left or right of an infix or postfix application has arity one, the parentheses may be omitted. Thus, we could also write

$$a@(b,c) \quad (a,b)@c$$

in the above example. Infix and postfix applications have lower precedence than the standard prefix application.

*Examples:* A standard situation of a function application is

$$+(\text{pred}(x), \text{succ}(y))$$

which can be made more readable by using the familiar infix notation

$$(\text{pred}(x)) + (\text{succ}(y))$$

and, if we use the binding rules, is even more readable

$$\text{pred}(x) + \text{succ}(y)$$

With higher-order functions, we obtain situations like

$$\text{filter}(\text{even})(S)$$

The application to the first argument list can also be written in postfix notation:

$$(\text{even filter})(S)$$

□

We employ a restricted infix notation to allow more than one infix application of one operator in sequence. For a given sequence of operators and operands during context analysis the compiler tries to deduce an unambiguous structure on operators and operands dependent on the functionality of the operator. If the deduced type is ambiguous then for the operators, which must be identical, right-associativity is assumed (universal right-associative rule).

*Example:* The functionality of prepending a data with a list and an infix notion of concat (here instantiated) would look like:

```
:: : nat ** seq[nat] -> seq[nat]
1 :: 2 :: 3 :: <>
```

The only type correct interpretation is:

```
1 :: ( 2 :: ( 3 :: <> ) )
```

The functionality of concatenating two lists and an infix notion of concat (again instantiated) would look like:

```

++ : seq[nat] ** seq[nat] -> seq[nat]
(1 :: <>) ++ (2 :: <>) ++ (3 :: <>)

```

Here, two bracketings are possible. But due to the universal right-associative rule the following bracketing is deduced:

```

(1 :: <>) ++ ((2 :: <>) ++ (3 :: <>))

```

□

### 3.4.4 Lambda Abstractions

The general form of lambda abstraction is

```

\\names . expression

```

where *names* is a (possibly empty) list of names for *lambda-bound variables*. Instead of a *name* a *wildcard* (`\_`) can be used to denote a placeholder for unused let-bound variables. The body of the abstraction is *expression*. The scope of the lambda-bound variables is the whole lambda abstraction.

The resulting expression is an unnamed function with the Cartesian product of the functionalities of the lambda-bound variables as its domain and the functionality of the body as its co-domain.

*Example:* A curried version of the addition function `+` from `Nat` can be written in the following way:

```

\\x.\\y.x+y

```

Its functionality is `nat->nat->nat`.

□

### 3.4.5 Sections

Sections are a short-hand notion for delayed evaluation of function applications. In an application arguments can be replaced by a section placeholder (`_`) to indicate the missing argument value. The behaviour of such an expression is explained best in giving the syntactical transformation: a lambda abstraction is built around the function application and for each placeholder a new lambda-bound variable is generated.

*Example:* An expression `a + _` is (syntactically) transformed into `\\ new. a + new` where `new` is a new variable. Multiple placeholders are possible, as in `f(_,a,b,_)` which is transformed into `\\ new1,new2. f(new1,a,b,new2)`. And an application like `f(_,a)(b,_)` is transformed into `\\ new1. \\ new2. f(new1,a)(b,new2)` □

### 3.4.6 Case Distinctions

The basis for case distinctions is the guarded expression, which is of the form

IF *guard* THEN *expression*

The *guard* is a boolean expression. The *expression* is only evaluated if the *guard* yields true. Its functionality is the functionality of the whole guarded expression.

Several guarded expressions can be assembled to form a *case distinction* of the form

IF *guard*<sub>1</sub> THEN *expression*<sub>1</sub>  
 ⋮  
 IF *guard*<sub>*n*</sub> THEN *expression*<sub>*n*</sub> FI

The operational semantics of the case distinction is defined as follows: The order in which the various guards are checked is left open (but fixed at compile time). When the first guard yielding true is encountered, its corresponding expression is evaluated. If none of the guards yields true, the case distinction is undefined (i.e., the program execution aborts).

*Note:* Each case distinction is *deterministic* in the sense that, once compiled, it is evaluated every time in the same order (even though this order is left to the compiler's discretion). For instance, we have the following property: the equation

$$\begin{aligned} f(x) == & \text{ IF } x \geq 0 \text{ THEN } g(x) \\ & \text{ IF } x < 0 \text{ THEN } h(x) \text{ FI} \end{aligned}$$

is equivalent to the operation

$$\begin{aligned} x > 0 & \Rightarrow f(x) \equiv g(x) \\ x < 0 & \Rightarrow f(x) \equiv h(x) \\ x = 0 & \Rightarrow f(x) \equiv g(x) \vee f(x) \equiv h(x) \end{aligned}$$

This equivalence characterizes the semantics of “nondeterminism” as realized by the OPAL compiler.

Sometimes we wish to evaluate certain guards *before* evaluating others. Then, we write

IF *guard*<sub>1</sub> THEN *expression*<sub>1</sub>  
 ⋮  
 IF *guard*<sub>*m*</sub> THEN *expression*<sub>*m*</sub>  
 OTHERWISE  
 IF *guard*<sub>*m*+1</sub> THEN *expression*<sub>*m*+1</sub>  
 ⋮  
 IF *guard*<sub>*n*</sub> THEN *expression*<sub>*n*</sub> FI

Here, the guards of the second block are only checked after all guards in the first block have yielded false. Of course, a case distinction can contain several sequential blocks separated by OTHERWISE.

Finally, we can conclude a case distinction — with or without OTHERWISE constructs — by an ELSE branch, which is only evaluated if none of the guards yields true.

```

IF guard1 THEN expression1
⋮
IF guardn THEN expressionn
ELSE expressionn+1 FI

```

In all cases, the keyword FI concludes the case distinction, thus allowing the unambiguous nesting of several case distinctions.

*Examples:* The lexicographic order  $\leq$  of texts is based on a case distinction like

```

DEF R<=S ==
    IF R empty?           THEN true
    OTHERWISE
    IF S empty?           THEN false
    OTHERWISE
    IF first(R)<first(S) THEN true
    IF first(R)>first(S) THEN false
                           ELSE rest(R)<=rest(S) FI

```

Our case distinctions also include the familiar IF-THEN-ELSE-FI construct as a special case:

```
IF x<0 THEN -(x) ELSE x FI
```

□

The *guards* can be sequentialized in using the keywords ANDIF or ORIF. These are only shortcuts for syntactical transformations as shown in the example.

*Example:*

```
IF guard1 ANDIF guard2 THEN expression1 ...
```

is transformed into

```
IF IF guard1 THEN guard2 ELSE false FI THEN expression1 ...
```

and

```
IF guard1 ORIF guard2 THEN expression1 ...
```

is transformed into

```
IF IF guard1 THEN true ELSE guard2 FI THEN expression1 ...
```

Sequential guards can be composed without bracketing. ANDIF and ORIF bind to the right. □

### 3.4.7 Extended Expressions

In order to allow a further structuring of expressions, it is possible to name the values of subexpressions and to use these names in other expressions. (This is particularly useful for “common subexpressions”.) In this way, we obtain *extended expressions* that comprise **LET** and **WHERE** clauses. The general notation is:

*expression* **WHERE** *declarations*  
**LET** *declarations* **IN** *expression*

Here, the *declarations* are sets of equations, the left-hand sides of which are (tuples of) names of *let-bound variables*. Thus, we have either of the two forms for a declaration:

*name* == *expression*  
(*name*<sub>1</sub>, ..., *name*<sub>*n*</sub>) == *expression*

Instead of *name* a *wildcard* can be used as a placeholder to denote unused values. The order in which these equations are written down is of no relevance, but cyclic dependencies are not allowed; in other words, it must be possible to put the declarations in a sequential order. (This is the compiler’s job.)

The scope of let-bound variables is the largest expression with which the **LET** or **WHERE** clause can be combined in a syntactically meaningful way. For instance, in **f(y)(y) WHERE y==...**, the scope does not only extend to the final (y), but (at least) to the whole term **f(y)(y)**. By contrast, in **f(y, y WHERE y==...)**, the scope only extends to the latter y. In addition, the scope of a let-bound variable extends over the whole list of declarations to which it belongs. The declared names inherit their functionalities from their corresponding right-hand sides.

The semantics of **LET** and **WHERE** clauses is *strict*, i.e., the equations are evaluated before the associated expression is evaluated.

*Examples:* In the translated equality test in section 3.3.2, we have a typical use of **LET** clauses.

Local declarations are often used to name the individual components of a function with several results:

**WHERE** (quotient,rest) == x divmod y

According to the scoping rules, the same name may, for example, appear in both branches of a conditional:

**IF** ... **THEN** **LET** xnew == f(xold) **IN** h(xnew,xold)  
                  **ELSE** **LET** xnew == g(xold) **IN** k(xnew,xold) **FI**

□

## Chapter 4

# Parameterization

Sequences of natural numbers are not much different from sequences of characters (= words) or even sequences of sequences of characters (= sentences). This fact is expressed — as it usually is in mathematics or informatics — by a proper parameterization.

### 4.1 Parameterized Structures

Below, we define the structure of sequences via a suitably chosen parameter, which in this case is a sort. However, we need not restrict this parameterization facility to sorts only. For instance, bounded sequences have an appropriate constant as their parameter, and ordered sequences are based on a suitable order relation on the element sort. The parameters are declared in the signature and can be used in the body of the structure. Thus, the scope within which the parameters are visible is the whole structure.

*Examples:* The structure `Seq` represents a classical example of parameterization. Here, `data` is the parameter, and it is declared to be a sort. Hence, it can be used in the structure body like any other sort.

```
SIGNATURE Seq[data]
-- Parameters
SORT data
-- Imports
IMPORT Nat ONLY nat
-- free type
TYPE seq == <>
           ::(ft:data, rt:seq)
-- additional operations
FUN ++ : seq**seq->seq          -- concatenation
FUN #  : seq->nat               -- length
-- etc.

IMPLEMENTATION Seq[data]
-- additional import
IMPORT Nat COMPLETELY
```



```

-- data definition
DATA seq == <>
           ::(ft:data, rt:seq)
-- concatenation
DEF <> ++ t2      == t2
DEF (c::t1) ++ t2 == c::(t1++t2)
-- length
DEF #(<>)      == 0
DEF #(_ :: t) == succ(#(t))
-- etc.

```

A structure for bounded sequences could be given in the form

```

SIGNATURE BoundedSeq [data,bound]
SORT data
FUN bound:nat
-- rest of signature part

```

Again, `data` can be used in the body like any normal sort, and `bound` can be used like any other constant. Note, however, that the sort `nat` must be known in the structure by means of an import, or it must be a parameter! (Using here a sort that is declared in the structure itself would clearly lead to a vicious circle.)

A structure for ordered sequences could be given in the form

```

SIGNATURE OrderedSeq [data,<]
SORT data
FUN < :data**data->bool
-- rest of signature part

```

As before, `data` can be used like a normal sort in the body, and `<` can be used like a normal operation. Note that a parameter which is a sort may be used to express the functionalities of other parameters.  $\square$

The rules for signatures, implementations and free types also apply to parameterized structures.

## 4.2 Instantiation

If a structure is parameterized, then all its constituents become *generic*. Every sort and operation now stands for a whole family of carrier sets, values and functions, respectively.

If we import a parameterized structure, we have to provide sorts and operations for the parameters. In case of overloaded imports, e.g., `Seq[nat]` and `Seq[real]`, we have to use additional facilities to make the names unequivocal (see section 5).

*Example:* We may now define the sort `text` from `Text` using the structure `Seq`.

```

IMPORT Seq[char] COMPLETELY
:
DATA text == text(text:seq, len:nat)

```

□

In the implementation part of a structure, it is also possible to import a parameterized structure *uninstantiated*. This can be viewed as an import of the structure with all necessary instantiations.

## Chapter 5

# Names and Scopes

In an OPAL program, there are many objects such as sorts, operations and parameters that must be referred to in the program text. Hence, all these objects have *names*. The use of each name is restricted to certain parts of the program text, called the scope of the name.

The naming facilities in OPAL are very flexible. This is practically unavoidable in connection with parameterization, but it is also convenient in other cases where overloading of identifiers is desirable. The classical example here is a function like addition on integers and real numbers.

### 5.1 Names

Names in OPAL do not only consist of an identifier, but also of an *origin* and a *kind*. For every name application, its origin and/or kind can be given in addition to the identifier in order to describe the name exactly. This is optional if the missing information can be deduced from the context.

#### 5.1.1 Origin

The origin of an object is the identifier of the structure in which it is declared.

*Example:* Suppose that there are two structures `DirGraph` and `UndirGraph` for directed and undirected graphs, respectively. Both sorts are called *graph*. To write a structure that uses both kinds of graphs, we have to import both structures:

```
IMPORT DirGraph    ONLY graph
IMPORT UndirGraph  ONLY graph
```

But we now encounter the problem of how to refer unequivocally to the sort of directed graphs. Hence, we may write `graph'DirGraph` (to be read as: “graph from `DirGraph`”) to refer to the sort `graph` of the structure `DirGraph`. □

Note that, in the case of transitive import, the origin is the structure in which the object is originally declared.

*Example:* The structure `Text` re-exports the sort `nat`'`Nat`. Thus, we can write

```
IMPORT Text ONLY nat
```

But the origin of `nat` remains the structure `Nat`. □

### 5.1.2 Kind

Since overloading is also allowed in the internal signature of a structure, the origin is not always sufficient to distinguish two objects. The *kind* of an object consists of its *class* (i.e., whether it is a sort or an operation) and — in case of an operation — its *functionality*.

*Example:* Suppose that, in a structure `Rat` for rational numbers, the sort as well as two conversion functions are all called `rat`; their identifiers are therefore overloaded:

```
SIGNATURE Rat
IMPORT Nat ONLY nat
IMPORT Int ONLY int
SORT rat
FUN rat:nat->rat
FUN rat:int->rat
```

To use one of these identifiers unambiguously, it is generally necessary to explicitly denote its kind. Thus, one has to write `rat:Sort`, `rat:nat->rat` and `rat:int->rat` in order to distinguish them from each other.

Note: If the kind can be deduced from the context, it can be omitted. For instance, it is sufficient to write `rat(-(1))`. □

Fortunately, the typing mechanism of OPAL is powerful enough to deduce the kinds in the majority of situations. Hence, the programs are not usually burdened with too many explicit denotations of origins and kinds. As a matter of fact, such annotations are the programmer's last resource on the few occasions where some ambiguity remains with respect to overloaded identifiers.

### 5.1.3 Instantiation

So far, we have looked at simple names. In case of parameterized structures, we have to distinguish between objects from different instantiations of a structure. Thus, the origin divides into the *origin identifier* and the (origin) *instantiation*. Either of these two parts can be omitted, if it is deducible from the context.

*Example:* If we import both `Seq[int]` and `Seq[real]`, we have to distinguish between the two instantiations of the sort `seq` by using their origins:

```
IMPORT Seq[int] ONLY seq
      Seq[real] ONLY seq
FUN intsToReals: seq'Seq[int]->seq'Seq[real]
```

If the origin identifier can be deduced from the context, we may leave it out:

```
FUN intsToReals:  seq[int]->seq[real]
```

□

### 5.1.4 Overload Resolution

In general, an object must be identified unequivocally. This is done by using the explicitly written parts of the name together with the information deduced from the context. If this “maximal” information still matches several visible names, we have a context error.

There is one exception to this rule: Where an overloaded identifier is imported, all its matching variations are imported. (If this is to be prohibited, one has to qualify the desired variant by appropriate annotations.)

*Example:* Suppose that the structure Seq defines four overloaded versions of the function ++, namely, concatenation of two sequences, of sequence and element, of element and sequence, and, finally, of two elements. If we wish to import only one of them, we may write

```
IMPORT Seq ONLY ++ :seq**seq->seq
```

□

## 5.2 Scopes

The scope of a name depends on the place of its declaration. In some cases, the scope can be extended and there may be *holes* in the scope.

### 5.2.1 Global Names in a Structure

The sorts and operations in the signature are *global names*. (Note that this includes the imported sorts and operations as well.)

- For the global names that are introduced in the *signature* part of a structure, the scope is the signature part and the implementation part.
- For the global names that are additionally introduced in the *implementation* part, the scope is only the implementation part.

There may be *holes* in the scope caused by local names (see section 5.2.2 below).

Global names may be declared several times; the repeated declarations introduce only one global name. (For instance, a repetition of the signature part in the implementation part would be legal; in practice, repeated declarations may occur through imports.)

### 5.2.2 Local Names

Pattern variables, lambda- and let-bound variables are *local names*. Local names consist of their identifiers and their kind only; there is no origin. The scope of a pattern variable is the function definition in which it is declared (see section 3.3); the scope of a lambda- or let-bound variable is the whole expression in which it is declared (see sections 3.4.4 and 3.4.7).

*Example:*

```

FUN quickSort:seq[nat]->seq[nat]
DEF quickSort(s) ==
  IF s <>? THEN <>
  IF s ::? THEN
    LET compare == ft(s)
    smaller == filter(\x.x<compare)(rt(s))
    equal    == filter(\y.y=compare)(s)
    larger   == filter(\z.z>compare)(rt(s))
  IN quickSort(smaller)++(equal++quickSort(larger))
FI

```

The pattern variable *s* can be used in the whole expression, whereas the let-bound variables *compare*, *smaller*, *equal* and *larger* can only be used inside the extended expression, and the lambda-bound variables *x*, *y* and *z* only in the corresponding lambda abstraction. □

Local names may cause holes in the scopes of global names. This happens where the local name has the same identifier as a global operation. In other words, there is no overloading between global and local names, except for sorts; their identifier may coincide with a local name.

*Example:* In the following definition, the function *length* cannot be used in the definition of *volume* because of its argument *length*.

```

FUN length: object->real
FUN volume: real**real**real->real
DEF volume(length,width,height) == length*width*height

```

□

For programming convenience, the definition of local names with the same identifier in overlapping scopes is forbidden.

*Examples:* In the following implementation, the let-bound variable *z* must be different from the pattern variables:

```

DEF f(x,y) == x*(IF y<0 THEN z WHERE z==y*y ELSE y FI)

```

However, the definition of two identical local names in non-overlapping (local) scopes is permitted:

```
DEF f(x) == IF x<0 THEN res WHERE res==x*neg(x)
           IF x>=0 THEN res WHERE res==x*x
           FI
```

In the case of pattern-based definitions, there may be identical pattern variables in different equations:

```
DEF <> ++s == s
DEF s++ <> == s
DEF (ft::rt)++s == ft::(rt++s)
```

□

## Chapter 6

# Programming in OPAL

This section describes some of the questions arising when running an OPAL program.

### 6.1 What is an OPAL Program?

An OPAL *program* consists of a *top-level structure* and the structures that are (transitively) imported by it. The import relation must be acyclic. For all structures constituting a program, a signature and an implementation part must exist. The top-level structure must export (at least) one special constant operation, the *top-level command* (see section 6.2).

An OPAL program can be compiled and linked together with a small runtime library to produce a self-contained, executable program. During the linking phase, the user is requested to specify the top-level command that is to be interpreted when the program is executed. This top-level command is interpreted in the runtime environment, which usually leads to a complete sequence of input/output actions.

### 6.2 Input/Output

OPAL realizes input/output by *commands* of the sort **com**. Typical -commands are requests such as “read the next input” and “write this afterwards”. Whenever a command is executed by the runtime system, it returns an answer of the sort **ans**. Both sorts are defined in the structure **Com**. Obviously, executing a program is a (complex) request to the runtime system. Therefore, the sort of the top-level command of an OPAL program must be of sort **com**.

The synchronization of commands and answers can be achieved through the introduction of continuations. A *continuation* simply encapsulates the synchronization between the user and the operating system. Commands can be built up by using the function `; provided` by the structure **ComCompose**. They have two “parts”: a simple command which the runtime system will interpret, and a continuation function that accepts the operating system answer and yields a new command. The runtime system will call this function after it has processed the request, and will pass as arguments the result of the request. Since the result of the continuation is a new command, the operating system will again interpret it, and so on.



This interface to I/O is monadic in the sense of [13, 16]. Computations are distinguished from values in that coomputations have parametrized type `com`. Computations which just return values are constructed by the (injection) function `yield`. Different computations are composed using the function `;` and these functions observe the laws defining monads. To our knowledge OPAL is the first programming language with an implemented monadic interface to I/O, since it was completed in this form in 1990.

### 6.2.1 A Simple Program

Our first program is an OPAL program that echoes the user's input until an empty line is entered. The structure `MyFirstProgram` is the top-level structure with `echo` as the top-level command:

```
SIGNATURE MyFirstProgram
IMPORT  Void          ONLY void
        Com[void]     ONLY com
FUN echo: com[void]           -- top-level command

IMPLEMENTATION MyFirstProgram
IMPORT  Void          ONLY void nil
        Nat           ONLY nat 0
        Char          ONLY char newline
        String        ONLY string empty?
        Com           ONLY com ans exit okay fail
        ComCompose    ONLY ;
        Stream        ONLY input stdin readLine
                        output stdout write
DEF echo == readLine(stdin) ; processline           -- (1)
FUN processline: ans[string]->com[void]
DEF processline(okay(s)) ==
  IF s empty? THEN exit(0)                          -- (2)
  ELSE write(stdout,s) ; (write(stdout,newline) ;    -- (3)
                        (readLine(stdin) ; processline)) FI -- (4)
DEF processline(fail(_)) ==
  write(stdout,"Cannot read user input")             -- (5)
```

The following aspects deserve mention:

Firstly, the `;` in line(3) is the standard case for a “write output” command that is followed by some other command ignoring the value returned by the runtime system. Here, `;` has the functionality

```
com[void]**com[void]->com[void]
```

Secondly, the `;` in line (1) with the functionality

```
com[string]**(ans[string]->com[void])->com[void]
```

is a typical way of building up a “read input” command followed by a function that processes its result. If `readLine:input->com[string]` is successfully evaluated, the

operating system returns the variant `okay` of the sort `ans[string]` that is fed up into the continuation function `processline`. In case an error results, the `fail` variant is returned.

Thirdly, line (2) and (5) are simple commands: the program will terminate.

### 6.2.2 State-Preserving Beyond Input/Output Boundaries

In general, commands may consist of input/output actions involving arbitrary complex data structures. State-preserving beyond input/output boundaries is realized by partial instantiation of functions that must still have the functionality `ans[...] -> com[...]` as their result functionality.

Our second program calculates the average of a sequence of numbers. Here, the first parameter group of the function `average` is used to transfer the count and sum of the already processed input. To shorten the presentation, we ignore structure boundaries and import clauses and do not give the implementations of `readNat` and `writeNat` (they are not part of the library).

```

FUN prog2: com[void]                                -- top-level command
DEF prog2 == (writeLine(stdOut,1stLn) ;
              writeLine(stdOut,2ndLn)) ;
              (prompt ; average(0,0))
WHERE 1stLn=="Average: Please type a sequence of numbers"
      2ndLn=="          0 stops input and prints the result:"
FUN prompt: com[nat]
DEF prompt == write(stdOut,"> ") ; readNat(stdIn)
FUN average: nat**nat->ans[nat]->com[void]
DEF average(count,sum)(okay(n)) ==                -- process correct input
  IF n=0 THEN write(stdOut,"= ") ;
              (writeNat(stdOut,sum/count) ;
               write(stdOut,newline))
              ELSE prompt ; average(count+1,sum+n) FI
DEF average(count,sum)(fail(_)) ==                -- ignore errors
  prompt ; average(count,sum)

```

### 6.2.3 Not Inherently Sequential Input/Output

With our fully referentially transparent commands, we are even able to cope with the organization of input that is not inherently sequential.

Suppose we wish to write a function `readLisp` that should substitute each atom of a lisp-like tree by a number that is to be interactively entered by the user. We define a command that preserves the shape of the given lisp-like structure in the following way (note that `o` is the function composition function, and that `&` is a variant of `;` that calls the continuation function only on success with the returned data):

```

DATA lisp == atom(valOf:nat)
              cons(car:lisp, cdr:lisp)
FUN readLisp: input**lisp->com[lisp]

```

```

DEF readLisp(in,atom(x)) ==
    readNat(in) & (yield o
                    (okay o
                     atom))
                                -- build a command that yields
                                -- the answer consisting of
                                -- an atom embedding
                                -- the result of readNat

DEF readLisp(in,lisp(l1,l2)) ==
    readLisp(in,l1) &
    (\\newl1.readLisp(in,l2) &
     (\\newL2.yield(okay(cons(newl1,newl2))))
    -- call readLisp of car
    -- then call readLisp of cdr
    -- then yield the new cons

```

### 6.3 The Library

With the exception of `bool`'`BOOL` and `denotation`'`DENOTATION`, OPAL has no sorts built into the language. Consequently, every data type that is to be used in a program has first to be defined by means of a suitable structure. It obviously makes no sense, though, to let every programmer start again from scratch. So there are some basic structures available in the programming environment of the OPAL compiler.

It should be noted, however, that these structures are not different from any other structure. In particular, most of them are written in OPAL. There are also some structures, though, that are substituted by hand-written ones, having the same semantics as an OPAL implementation, but more efficient in terms of storage and/or time. (The OPAL user is not aware of the kind of structure he uses.)

The structure `Seq`, for example, contains at least those functions that are presented in section 4.1. A detailed description of the library is given in "Bibliotheca Opalica - A Document on Structured Use and Abuse", which is available in the distribution of the OPAL system.

# Appendix A

## OPAL Syntax

The formal definition of the syntax is given in EBNF. For each grammar rule, the corresponding context conditions and attributes are specified informally. The context conditions are preceded by ‘@’, and the attributes by ‘•’. Syntactic transformations are preceded by  $\Downarrow$ . The context conditions are additionally supplied with an underlined short name reflecting the corresponding error. Some conditions that are considered by the identification function are double-underlined. For convenience’s sake the grammar rules are presented in semantic sections, each beginning with a short description. Furthermore, they are formulated with a view to providing good readability for humans rather than facilitating parsing.

The metasympols of EBNF are set as  $[, ], (, ), |, ||, ^+$  and  $*$ ; nonterminals are set as exemplified by *SignaturePart* and terminals as “IMPORT”.

The other extensions to normal BNF have the following meanings (see [17]):

<i>Abbreviation</i>	<i>Meaning</i>
$X ::= \alpha(\beta)\gamma$	$X ::= \alpha Y \gamma$ $Y ::= \beta$
$X ::= \alpha[\beta]\gamma$	$X ::= \alpha\gamma   \alpha(\beta)\gamma$
$X ::= \alpha u^+ \gamma$	$X ::= \alpha Y \gamma$ $Y ::= u   Y u$
$X ::= \alpha u^* \gamma$	$X ::= \alpha[u^+] \gamma$
$X ::= \alpha    t$	$X ::= \alpha(t\alpha)^*$

Here,  $\alpha$ ,  $\beta$  and  $\gamma$  are arbitrary right-hand sides of rules,  $Y$  is a new nonterminal,  $u$  is either a single symbol or a parenthesized right-hand side, and  $t$  is a terminal symbol.

### A.1 Definitions and general context conditions

*Names* are used to refer to objects in a structure. To distinguish overloaded objects, names have two components besides the *identifier*: the origin and the kind. The *origin*

reflects the (instance of a) structure in which the name is declared; the *origin identifier* is the name of this structure, the *instantiation* is a list of names (called *instance names*) referring to the formal or actual parameters of the structure. The *kind* of a name distinguishes sorts and operations, and for operations it reflects the operation's functionality based on the names of the used sorts.

There are two kinds of names, depending on their scope. *Global names* refer to sorts and operations of a structure. A global name can be given *attributes* indicating that it is the  $n^{\text{th}}$  parameter of a structure, it is a *free constructor* and its  $n^{\text{th}}$  instance name must be *known*. *Local names* refer to pattern variables, lambda-bound and let-bound variables, all of which are operations; they consist only of an identifier and a kind. There are three general context conditions for names:

- © local name as actual parameter: The instantiation of a name must be a list of global names.
- © compound object: The functionality of an operation's name may not be a Cartesian product.
- © local sort: A local name may not be a sort.

A *signature* is a set of names. The *exported signature* of a structure contains the names that can be imported by other structures. The *global signature* of a structure (or its signature or implementation part) consists of the global names of the structure (or its parts). It is divided into two disjoint sets: the *internal signature* contains the names that are *declared* by an explicit declaration; the *external signature* contains the *imported* names. The same global name can be declared (or imported) twice.

A global name must be complete in order to be used. A global name is *complete* if all the names in its constituent parts are elements of the global signature. For this, it is sufficient to demand that a name of the global signature is semicomplete (and this is necessary for uninstantiated imports). A global name is *semicomplete* if all names in its constituent parts are elements of the global signature or *unknown* instance names. There are two general context conditions for the names of the global signature:

- © incomplete name: All names of the global signature must be semicomplete.
- © recursive name: No name of the global signature may contain itself unless it is a parameter.

Function definitions and their parts have a *local signature*, which is the set of global and local names that can be used in them. Each global name of the global signature is contained in a local signature if it is a sort, or there is no local name with the same identifier in this local signature. A local name is contained in a local signature if it is contained in the local signature of the surrounding construct, or it is explicitly *added* to the local signature. There are two general context conditions for local signatures:

- © overloaded local name: There may not be two local names with the same identifier in a local signature.

- © duplicate local name: A local name may not be added twice to a signature.

A *substitution* with respect to a list of *formal* names, and a list of *actual* names of the same length, is a function that yields for a name the original name with all occurrences of the formal names replaced by the corresponding actual names. If a substitution replaces the  $n^{\text{th}}$  instance name of the original name, the  $n^{\text{th}}$  instance name of the yielded name must be *known*. In case of a *free* substitution, the  $n^{\text{th}}$  instance name must not necessarily be known.

A substitution is *proper* if both lists have the same length and the substitution of the kind of each formal name yields the kind of the corresponding actual name. A substitution of a signature is the set of substitutions of all its elements.

In the syntax, the denotation of a name may omit parts of it. Such a denotation is called a *partial name*. A partial name *matches* a name if all its given parts match. If parts of an origin are given, the partial name does not match any local name. The *matching set* of a partial name is the set of matching names in the global or local signature under consideration.

A *selection* is a function which maps every partial name (except those listed in selected imports) to an element of its matching set, called the *selected name*. A selection is *more specific* than another if they only differ at points where the first yields a global name and the other a local name. An *identification* is a selection, where all double-underlined context conditions are fulfilled and there is no more specific selection that considers these conditions. The selected name for an identification is called the *identified name*. There are two major context conditions:

- © undefined identification: An identification must exist, i.e., there is a function from partial names to names so that all double-underlined context conditions are fulfilled for the identified names.
- © ambiguous identification: The identification must be unique, i.e., there is a most specific function from partial names to names so that all double-underlined context conditions are fulfilled.

An *application analysis* is a function which takes a sequence of expressions and assigns each operator its arguments called *identified arguments*. An argument is an *identified argument* iff it is type correct regarding the functionality of the operator. If the analysis can find an unambiguous and type correct application structure for all expressions of the sequence it delivers the deduced structure with one operator as *top operator* of the structure. If the analyzed structure is ambiguous the analysis tries to build an unambiguous application considering right-associativity for the operator. An error is yielded if no unambiguous application regarding the types and the right-associative rule can be deduced.

Each name of the internal signature (except the parameters) can be implemented. An *implementation* of a name is a syntactical part of the program text such as a data type definition or a set of function definitions. There are two general context conditions:

- © parameter implementation: There may not be an implementation of a parameter.
- © duplicate implementation: There may not be more than one implementation of a name.

## A.2 Structures

A program is a collection of structures. A structure consists of a signature part and an implementation part. The former is the export interface of the structure and can be compiled separately from the latter.

- © duplicate structure name: Every structure in a program must have a unique identifier.

$$\begin{aligned} \text{SignaturePart} ::= & \text{“SIGNATURE” } \text{Ident} [ \text{“[” } \text{Name} \parallel \text{“,” } \text{“]”} ] \\ & ( \text{Signature} \mid \text{FreeType} )^* \end{aligned}$$

- The global signature of the signature part consists of all names that are declared or imported in the signature part.
- The  $n^{\text{th}}$  identified name of the optional *Name* list is the  $n^{\text{th}}$  parameter.
- © duplicate parameter: The identified names must be different.
- © imported parameter: The identified names must be elements of the internal signature.
- © invisible parameter kind: The kind of the identified names may not contain names of the internal signature unless they are parameters themselves.
- The actual origin of the structure consists of *Ident* as origin identifier, and the list of identified names as instantiation.
- The exported signature of the structure is the signature of the signature part without the parameters.
- © empty export: The exported signature may not contain only names with the origin “BOOL” or “DENOTATION”.
- © possible name clash: The internal signature of the signature part may not contain two names for which there are actual parameters, so that their substitutions with respect to the formal and actual parameters yield the same name.

$$\begin{aligned} \text{ImplementationPart} ::= & \text{“IMPLEMENTATION” } \text{Ident} [ \text{“[” } \text{Name} \parallel \text{“,” } \text{“]”} ] \\ & ( \text{Signature} \mid \text{FreeType} \\ & \mid \text{DataDefinition} \mid \text{FunDefinition} )^* \end{aligned}$$

- © no signature part: There must be a signature part with *Ident* as structure identifier.
- The global signature of the implementation part consists of the global signature of the corresponding signature part (with all free constructor attributes reset) together with all names that are declared or imported in the implementation part.

- © different parameter list: If a *Name* list is given, the  $n^{\text{th}}$  identified name must be the  $n^{\text{th}}$  parameter, and all parameters must be given.

### A.3 Signature

The signature provides sorts and operations of the internal and external signature. Repeated declarations and imports are possible.

$\text{Signature} ::= \text{"SORT"} \text{ Ident}^+ \mid \text{"FUN"} ( \text{Ident}^+ \text{ ":" } \text{Functionality} )^+ \mid \text{Import} \mid \text{Law}$

- In case of a “SORT” declaration, for each *Ident* a name with the given *Ident* as identifier, the actual origin as origin, and “SORT” as its kind is declared to be element of the internal signature.
- In case of a “FUN” declaration, for each *Ident* a name with the given *Ident* as identifier, the actual origin as origin, and the given functionality as its operation’s kind is declared to be element of the internal signature.

$\text{Import} ::= \text{"IMPORT"} ( \text{Ident} [ \text{"[" } \text{Name} \parallel \text{" , " } \text{ "]" } ] \text{Selection} )^+$

- The structures “BOOL” and “DENOTATION” are always imported completely.
- © unknown structure: *Ident* must be the identifier of a structure in the program.
- © cyclic import: The structures must be in an acyclic import relation. (In particular, *Ident* may not be the identifier of the actual structure.)
- © uninstantiated import in signature part: If an import of a parameterized structure occurs in a signature part, a *Name* list must be given.
- The identified names of the optional *Name* list are called actual parameters.
- © improperly instantiated import: If a *Name* list is given, the substitution with respect to the formal and actual parameter lists must be proper.
- If a *Name* list is given, all names of the structure’s exported signature are substituted with respect to the formal and actual parameters. The substituted names constitute the import’s signature. If no list is given, the union of all proper free substitutions of the names of the structure’s export signature is the import’s signature.

$\text{Selection} ::= \text{"ONLY"} \text{ Name}^+ \mid \text{"COMPLETELY"}$

- In case of an “ONLY” import, for each *Name* all matching names of the import’s signature are imported.



- © empty import: In an “ONLY” import, each *Name* must match at least one name of the import’s signature.
- © instantiation in selection: If *Name* matches one substitution of an exported name in an uninstantiated import, it must match all its other proper substitutions, too.
  - In case of a “COMPLETELY” import, all names of the import’s signature are imported.

*Functionality* ::= *ProductFct* | *FunctionFct*

*ProductFct* ::= *SimpleFct* || “\*\*” | “(” *ProductFct* “)”

*FunctionFct* ::= *ProductFct* “->” *Functionality* | “(” “)” “->” *Functionality*

*SimpleFct* ::= *Name* | “(” *FunctionFct* “)” | “(” *SimpleFct* “)”

- © operation as sort: The identified name must be a sort.

## A.4 Free Types and Data Definitions

Sorts are implemented by data definitions and can be specified by free types.

*FreeType* ::= “TYPE” *Name* “==” *Variant*<sup>+</sup>

- The name consisting of the identifier of the partial name *Name*, the actual origin, and “SORT” as its kind is declared to be element of the internal signature.
- © improperly named free type: *Name* must match the declared name.
- © parameter as free type: *Name* may not be a parameter.
- © duplicate free type: There may not be more than one *FreeType* for the declared name.
  - All constructors that are declared in the *Variant* list are free constructors.
- © duplicate free constructor/discriminator: All constructors and discriminators that are declared in the *Variant* list must be different.
- © parameter as free constructor/discriminator/selector: No constructor, discriminator or selector may be a parameter.

*DataDefinition* ::= “DATA” *Name* “==” *Variant*<sup>+</sup>

- The name consisting of the identifier of the partial name *Name*, the actual origin, and “SORT” as its kind is declared to be element of the internal signature.

- *DataDefinition* is an implementation of the declared name and of all selectors of the *Variant* list.
- improperly named data definition: *Name* must match the declared name.
- If there is no free type for the declared name, all constructors that are declared in the *Variant* list are said to be free constructors.

*Variant* ::= *Name* [ *Components* ]

- The name consisting of the identifier of the partial name *Name*, the actual origin, and the “constructor functionality” as its kind is declared to be an element of the internal signature. If *Components* are given, the constructor functionality is the function functionality with the functionality of *Components* as its domain and the implemented (or specified) sort as its co-domain, otherwise the constructor functionality is the implemented sort. The declared name is called constructor.
- © improperly named constructor: *Name* must match the constructor.
- The name consisting of the identifier of the partial name *Name* with an appended question mark, the actual origin, and the “discriminator functionality” as its kind is declared to be an element of the internal signature. The discriminator functionality is the function functionality with the implemented (or specified) sort as its domain and the predefined sort “bool” as its co-domain. The declared name is called discriminator.
  - *Variant* is an implementation of the constructor and of the discriminator, if it is part of a *SortImplementation*.

*Components* ::= “(” ( *Name* “:” *Functionality* ) || “,” “)”

- For each *Name*, the name consisting of the identifier of the partial name *Name*, the actual origin, and its “selector functionality” as its kind is declared to be an element of the internal signature. The selector functionality is the function functionality with the implemented (or specified) sort as its domain and the given functionality as its co-domain. The declared name is called selector.
- © improperly named selector: *Name* must match the declared name.
- © duplicate selector definition: All selectors must be different.
- © tupled component: No *Functionality* may be a Cartesian product.
- The functionality of *Components* is the Cartesian product of all functionalities.

## A.5 Function Definitions

Operations are implemented by a set of function definitions. The left-hand side of a function definition determines the definition target. There are three different kinds of expressions on the left-hand side that are similar but differ in details: complete left-hand sides, patterns and pattern constructors. *Complete left-hand sides* are expressions that contain the definition target, which is applied to *patterns*. A pattern is a *pattern constructor* applied to patterns or a pattern variable, or a pattern constructor. To shorten the syntax description, they are only distinguished by the additional attribute *expression kind*.

- The set of *FunImplementation* with the same definition target is an implementation of the definition target.

*FunDefinition* ::= “DEF” *TopLeftHandSide* “==” *Expression*

- *LeftHandSide* is a complete left-hand side.
- The pattern variables of *LeftHandSide* are added to the local signature of the *FunDefinition*.
- © wrongly typed implementation: The functionality of *LeftHandSide* must equal *Expression*.

*TopLeftHandSide* ::= *SimpleLhs* | *LhsTopInfix*

*SimpleLhs* ::= *LhsName* | *LhsTuple* | *LhsApply* | *Wildcard*

- If *SimpleLhs* is a *Wildcard* then *SimpleLhs* is a pattern.

*LhsTopInfix* ::= *SimpleLhs* *LhsName* [ *SimpleLhs* ] | “(” “)” *LhsName*

- © higher-order infix pattern constructor: *LhsTopInfix* may not be a higher-order pattern constructor.
- Both *SimpleLhs* are patterns. *LhsName* is a pattern constructor if *LhsTopInfix* is a pattern, and it is complete if *LhsTopInfix* is complete.
- © wrongly typed lhs infix: The functionality of *LhsName* must be a function functionality with the (flattened) Cartesian product of the functionalities of the given *SimpleLhs* (or the empty Cartesian product) as its domain.
- The functionality of *LhsTopInfix* is the co-domain of the functionality of *LhsName*.

$$\begin{aligned}
LeftHandSide &::= SimpleLhs \mid LhsInfix \\
LhsInfix &::= SimpleLhs \ LhsName \\
&\quad \mid SimpleLhs \ ( \ LhsName \ SimplLhs \ )^+ \\
&\quad \mid \text{“}(\text{“} \text{”} \text{”} \ LhsName
\end{aligned}$$

- © higher-order infix pattern constructor: *LhsInfix* may not be a higher-order pattern constructor.
- All *SimpleLhs* are patterns. *LhsName* is a pattern constructor if *LhsInfix* is a pattern, and it is complete if *LhsInfix* is complete.
- All *LhsName* must be the same complete name.
- The binding of *LhsName* is right-associative.
- © wrongly typed lhs infix: The functionality of *LhsName* must be a function functionality with the (flattened) Cartesian product of the functionalities of the given *SimpleLhs* (or the empty Cartesian product) as its domain.
- The functionality of *LhsInfix* is the co-domain of the functionality of *LhsName*.

$$\begin{aligned}
LhsTuple &::= \text{“}(\text{“} \ [ \ LocalName \ \text{“AS”} \ ] \ LeftHandSide \ ) \ || \ \text{“},\text{“} \ \text{“} \ \text{”} \\
&\quad [ \ \text{“:”} \ Functionality \ ]
\end{aligned}$$

- Each *LeftHandSide* inherits its expression kind from *LhsTuple*.
- © improper tuple: If *LhsTuple* is not a pattern, it must not contain more than one *LeftHandSide*.
- © improper pattern synonym: If *LocalName* is given, *LeftHandSide* must be a pattern.
- The identified names for every *LocalName* are pattern variables.
- © wrongly typed pattern synonym: The functionality of a pattern variable and the functionality of the corresponding *LeftHandSide* must be equal.
- The functionality of *LhsTuple* is the (flattened) Cartesian product of the functionalities of all *LeftHandSides*.
- © wrong typing of lhs tuple: The functionality of *LhsTuple* must equal a given *Functionality*.

$$LhsApply ::= SimpleLhs \ ( \ LhsTuple \ | \ \text{“}(\text{“} \ \text{”} \ \text{”} \ )$$

- © higher-order pattern constructor: *LhsApply* may not be a higher-order pattern constructor.

- *LhsTuple* is a pattern. *SimpleLhs* is a pattern constructor if *LhsApply* is a pattern, and it is complete if *LhsApply* is complete.
- © wrongly typed lhs apply: The functionality of *SimpleLhs* must be a function functionality with the functionality of the given *LhsTuple* (or the empty Cartesian product) as its domain.
- The functionality of *LhsApply* is the co-domain of the functionality of *SimpleLhs*.

*LhsName* ::= *Name*

The context conditions and attributes for *LhsName* differ for complete expressions, patterns and pattern constructors. They are therefore listed separately.

If *LhsName* is a complete left-hand side:

- The identified name is the definition target.
- © improperly named function: The identified name must be an element of the internal signature.
- © sort defined by function definition: The identified name must be an operation.
- The functionality of *LhsName* is the functionality of the identified name.

If *LhsName* is a pattern constructor:

- © non-constructor used as constructor: The identified name must be a free constructor.
- The functionality of *LhsName* is the functionality of the identified name.

If *LhsName* is a pattern, there are two possibilities:

- If the identified name is a local name, it is a pattern variable.
- © non-constant constructor used as pattern: If the identified name is a global name, it must be a free constructor with a sort as its functionality.
- The functionality of *LhsName* is the functionality of the identified name.

## A.6 Expressions

Expressions are used to define the right-hand side of function definitions.

$$\begin{aligned}
 \textit{Expression} & ::= \textit{SimpleExpression} \mid \textit{Infix} \mid \textit{Abstraction} \\
 & \quad \mid \textit{Cases} \mid \textit{Let} \mid \textit{Where} \\
 \textit{SimpleExpression} & ::= \textit{ValueDenotation} \mid \textit{Tuple} \mid \textit{Application} \\
 \textit{ValueDenotation} & ::= \textit{Name} \mid \textit{Denotation} \mid \textit{SectionPattern}
 \end{aligned}$$

© sort as operation: The identified name must be an operation.

- The functionality of *Name* is the functionality of the identified name, the functionality of *Denotation* is the predefined sort “denotation”. The functionality of *ValueDenotation* is the functionality of the given alternative.

$Tuple ::= “(” Expression \parallel “,” “)” [ “:” Functionality ]$

- The functionality of *Tuple* is the (flattened) Cartesian product of the functionalities of all *Expressions*.

© wrong typing of tuple: The functionality of *Tuple* and a given *Functionality* must be equal.

$Application ::= SimpleExpression ( Tuple \mid “(” “)” )$

© wrongly typed application: The functionality of *SimpleExpression* must be a function with the functionality of the given *Tuple* (or the empty Cartesian product) as its domain.

↓ If *Tuple* contains *SectionPattern* then *Application* is transformed as follows:

$$\begin{aligned} & SimpleExpression ( e_1, e_2, \dots, e_n ) \wedge \exists e_i \xrightarrow{*} SectionPattern \\ & \quad \downarrow \\ & \forall e_j \dots e_k \xrightarrow{*} SectionPattern \wedge ne_j \dots ne_k \text{ are new variables } \wedge \\ & \quad \setminus ne_j, \dots, ne_k . SimpleExpression ( e_1, e_2, \dots, ne_j, \dots, ne_k, \dots, e_n ) \end{aligned}$$

- The functionality of non-terminal *Application* is the co-domain of the functionality of *SimpleExpression*.

$Infix ::= SimpleExpression Name$   
 $\mid SimpleExpression ( Name SimpleExpression )^+$   
 $“(” “)” Name$

© sort as infix operation: All identified names must be an operation.

- All identified names must be the same name.

© wrongly typed infix: The functionality of the identified name must be a function functionality with the (flattened) Cartesian product of the functionalities of the given *SimpleExpression* (or the empty Cartesian product) as its domain.

- The functionality of *Infix* is the co-domain of the functionality of the identified *top operator*.

$Abstraction ::= "\backslash" [ LocalName \parallel ", " ] "." Expression$

Since *Abstraction* has a higher priority than “WHERE” expressions, *Expression* cannot be a “WHERE” expression.

- The identified names for every *LocalName* are lambda-bound variables.
- The lambda-bound variables are added to the local signature of *Abstraction*.
- The functionality of *Abstraction* is a function functionality with the Cartesian product of the functionalities of the lambda-bound variables as its domain and the functionality of *Expression* as its co-domain.

$Cases ::= Guard^+ \parallel \text{“OTHERWISE”} [ \text{“ELSE”} Expression ] \text{“FI”} [ \text{“:”} Functionality ]$

- © incompatible guards: The functionalities of each *Guard* must be equal.
  - The functionality of *Cases* is the functionality of the first *Guard*.
- © incompatible else: The functionality of *Cases* and the functionality of a given *Expression* must be equal.
- © wrong typing of cases: The functionality of *Cases* and a given *Functionality* must be equal.

$Guard ::= \text{“IF”} Expression \parallel ( \text{“ANDIF”} \mid \text{“ORIF”} ) \text{“THEN”} Expression$

↓

$$\begin{array}{c} Expression_1 \text{“ANDIF”} Expression_2 \\ \downarrow \\ \text{“IF” } Expression_1 \text{“THEN” } Expression_2 \text{“ELSE” false “FI”} \end{array}$$

and

$$\begin{array}{c} Expression_1 \text{“ORIF”} Expression_2 \\ \downarrow \\ \text{“IF” } Expression_1 \text{“THEN” true “ELSE” } Expression_2 \text{“FI”} \end{array}$$

- “ANDIF” and “ORIF” are right-associative.
- The functionality of *Guard* is the functionality of the second expression.
- © wrongly typed condition: The functionality of the first expression must be the predefined sort “bool”.

$Let ::= \text{“LET” } Equation^+ \text{ “IN” } Expression$

Owing to the binding rules of “WHERE” expressions, only the last *Equation* may have a “WHERE” expression as its right-hand side. Since “LET” has a higher priority than “WHERE”, *Expression* cannot be a “WHERE” expression.

- The let-bound variables of the *Equation* list are added to the local signature of *Let*.
- © recursive let: There must exist an order of all *Equations*, so that every *Equation* in which a let-bound variable is used on its right-hand side is preceded by the *Equation* in which this variable is used on the left-hand side.
- The functionality of *Let* is the functionality of *Expression*.

$Where ::= Expression \text{ “WHERE” } Equation^+$

All the following equations bind to the “WHERE” expression.

- The let-bound variables of the *Equation* list are added to the local signature of *Where*.
- © recursive where: There must exist an order of all *Equations*, so that every *Equation* in which a let-bound variable is used on its right-hand side is preceded by the *Equation* in which this variable is used on the left-hand side.
- The functionality of *Where* is the functionality of *Expression*.

$Equation ::= (LocalName \mid \text{“(” } LocalName \parallel \text{“,” “)” } ) \text{ “=” } Expression$

- The identified names for every *LocalName* are let-bound variables.
- © wrongly typed equation: The Cartesian product of the functionalities of the let-bound variables and the functionality of *Expression* must be equal.

### A.6.1 Bracketing of Infix-Expressions

The OPAL compiler analyzes infix expressions in the context checking phase. The compiler uses all available information for this task. However, there are situations, where there remain ambiguities, for example in parsing arithmetic expression.

Let  $a \oplus b \otimes c$  the infix expression to be analyzed, and suppose that both possible bracketings,  $(a \oplus b) \otimes c$  and  $a \oplus (b \otimes c)$  are type correct.

The OPAL compiler maintains two relations on functions,  $\mathcal{L}$  and  $\mathcal{R}$ . If  $(\oplus, \otimes) \in \mathcal{L}$ , the first possibility is chosen, if  $(\oplus, \otimes) \in \mathcal{R}$ , the second possibility is chosen. If  $(\oplus, \otimes) \notin (\mathcal{L} \cup \mathcal{R})$ , and both functions are equal, right bracketing is chosen as default.

Defining bracketings for other infix expressions is not yet part of the language, but may be defined with pragmas. Function pairs are inserted into  $\mathcal{L}$  or  $\mathcal{R}$  by means of the following pragmas:



© wrongly typed equality: The functionality of both *Expressions* in an *Equality* or *Inequality* must be equal. If only one *Expression* is given, its functionality must be the predefined sort “bool”.

$$\begin{aligned}
\textit{PropositionalFormula} &::= \textit{Negation} \mid \textit{Conjunction} \\
&\quad \mid \textit{Disjunction} \mid \textit{Implication} \mid \textit{Equivalence} \\
\textit{Negation} &::= \text{“NOT” } \textit{Formula} \\
\textit{Conjunction} &::= \textit{Formula} \text{ “AND” } \textit{Formula} \\
\textit{Disjunction} &::= \textit{Formula} \text{ “OR” } \textit{Formula} \\
\textit{Implication} &::= \textit{Formula} \text{ “==>” } \textit{Formula} \\
\textit{Equivalence} &::= \textit{Formula} \text{ “<=>” } \textit{Formula}
\end{aligned}$$

The rules for *PropositionalFormula* are ordered in descending priority and right-associativity is assumed for equal formulas. Since *QuantifiedFormula* has an even less priority the first formula of *Conjunction*, *Disjunction*, *Implication* and *Equivalence* cannot contain an unbracketed *QuantifiedFormula*.

$$\textit{QuantifiedFormula} ::= ( \text{“ALL”} \mid \text{“EX”} ) \textit{LocalName}^+ \text{“.”} \textit{Formula}$$

- © The identified names for every *LocalName* are quantor-bound variables. They are added to the local signature of *QuantifiedFormula*.

## A.8 Partial Names

Partial names are the syntactical means for referring to names of the signature under consideration.

$$\textit{Name} ::= \textit{Ident} [ \text{“,”} \textit{Ident} ] [ \text{“[”} \textit{Name} \parallel \text{“,”} \text{“]”} ] [ \textit{Kind} ]$$

- *Name* is a partial name with the first *Ident* as identifier and, if given, the second *Ident* as origin identifier, the list of identified names as instantiation, and *Kind* as kind.

$$\textit{Kind} ::= \text{“:”} \text{“SORT”} \mid \text{“:”} \textit{Functionality}$$

$$\textit{LocalName} ::= ( \textit{Ident} \mid \textit{Wildcard} ) [ \text{“:”} \textit{Functionality} ]$$

- *LocalName* is a partial name with *Ident* as identifier and, if given, *Functionality* as kind.

$$\textit{Wildcard} ::= \text{“_”}$$

- Wildcard is a fresh local name.

$$\textit{SectionPattern} ::= \text{“_”}$$

- SectionPattern is a fresh local name.

$Ident ::= Ide \ [ \ Pragma \ ]$

## A.9 Lexical Rules

The syntax description treats the program text as a sequence of *lexemes*, i.e., keywords and separators (denoted by their character representation), identifiers, denotations and pragmas. Comments and layout are ignored. If there is no way to partition a program text into a lexeme sequence, then there is a lexical error. The description of the lexeme sequence itself treats the program text as a sequence of *symbols*, which are ultimately defined on the basis of characters. The different kinds of lexemes and symbols are described as sets of symbol sequences (or character sequences). Therefore, the usual set operations (union, set difference) are used together with sequencing operations (concatenation, repetition). Sets are built from elements by enclosing the collection of elements (without any separating comma) in curled brackets.

### Lexemes

The lexemes are defined on the basis of symbols, sometimes denoted by their character representation. There are seven kinds of lexemes: keywords, identifiers, denotations, separators, comments, pragmas and layout.

<i>Lexeme</i>	=	$Keyword \cup Ide \cup Denotation \cup Separator \cup$ $Comment \cup Pragma \cup Layout$
<i>Keyword</i>	=	{ ALL AND ANDIF AS COMPLETELY DATA DEF DFD ELSE EX FI FUN IF IMPLEMENTATION IMPORT IN LAW LET NOT ONLY OR ORIF OTHERWISE SIGNATURE SORT THEN TYPE WHERE ** -> . : == _ === <<= ==> <=> \\ }
<i>Ide</i>	=	$( Alphanum \cup Graphic ) \setminus ( Keyword \cup \{ -- /* */ /\$ \$/ \} )$
<i>Denotation</i>	=	<i>String</i>
<i>Separator</i>	=	<i>Delimiter</i>
<i>Comment</i>	=	-- ( <i>Symbol</i> \ { nl } ) <sup>*</sup> nl $\cup$ /* ( <i>Symbol</i> \ { -- /* */ } $\cup$ <i>Comment</i> ) */
<i>Pragma</i>	=	/\$ ( <i>Lexeme</i> \ <i>Pragma</i> ) <sup>*</sup> \$/
<i>Layout</i>	=	<i>Blank</i> <sup>+</sup>

Note that keywords are reserved words, they cannot be used as identifiers, except for the keyword . (single dot), which is only recognized after one of the keywords \\ ALL EX. Note also that there are two kinds of comments: a *line comment* starts with the symbol -- and skips all symbols (and therefore all characters) until the next newline; a *nested comment* is enclosed within the symbols /\* and \*/ and may again contain comments. Line comments have a higher priority than nested comments, i.e., the nested comment symbols /\* and \*/ inside a line comment are ignored.

## Symbols

There are six kinds of symbols: alphanumeric and graphic symbols, delimiters and blanks, strings and wrong symbols. Note that wrong symbols can occur in correct programs inside comments.

<i>Symbol</i>	=	$Alphanumeric \cup Graphic \cup Delimiter \cup Blank \cup String \cup Wrong$
<i>Alphanumeric</i>	=	$(Letgit \cup \_ )^+ ?^*(\_ (Alphanumeric \cup Graphic))^*$
<i>Graphic</i>	=	$Special^+ (\_ (Alphanumeric \cup Graphic))^*$
<i>Delimiter</i>	=	$Extra \setminus \{ " \}$
<i>Blank</i>	=	$White$
<i>String</i>	=	$" (Char \setminus (Other \cup \{ " \text{ tab nl } \})) \cup "" \cup Escape)^* "$
<i>Wrong</i>	=	$Other \cup$ $" (Char \setminus (Other \cup \{ " \text{ tab nl } \})) \cup "" )^*$

Strings are enclosed in quotes and may contain double quotes; tabulators, newlines, and non-printable characters cannot be used in strings, so a string can never extend over more than one line.

The following escape sequences are recognized within strings and replaced by one character.

<i>Escape</i>	=	{	$\backslash a$	(alarm)
			$\backslash b$	(backspace)
			$\backslash f$	(formfeed)
			$\backslash r$	(carriage return)
			$\backslash t$	(tabulator)
			$\backslash v$	(vertical tab)
			$\backslash \backslash$	(single backslash)
			$\backslash ?$	(questionmark)
			$\backslash ' $	(single quote)
			$\backslash "$	(double quote)
			$\backslash oct [ oct [ oct ] ]$	
			$\backslash x hex^+$	
hex	=	{	0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F }	
oct	=	{	0 1 2 3 4 5 6 7 }	

© number too large The octal or hexadecimal number may not exceed 255.

Wrong symbols are “incorrect strings” or non-printable and non-ASCII characters. The partitioning of a program text into a symbol sequence  $p$  must obey the condition longest match: for all  $l, r \in Symbol^*$  and  $u, v \in Symbol$  with  $p = luvr$ , the following implications must hold ( $v_1$  denotes the first character of  $v$ ):

$u \in Letgit^+$	$v_1 \notin (Letgit \cup \{ ? \})$
$u \in Letgit^+ ?^+$	$v_1 \neq ?$
$u \in Graphic$	$v_1 \notin Special$
$u \in String$	$v_1 \neq "$
$u \in Wrong \setminus Other$	$v_1 \notin Other \cup \{ \text{tab nl} \}$

## Unicode Escapes

The scanner recognizes unicode escapes.

$UnicodeEsc = "\text{“u”}^+ hex\ hex\ hex\ hex$

The unicode escape is replaced by the unicode character with the code number represented by the four following hexadecimal digits.

Note that unicode escapes are processed before the proper scanner starts.

- © The last “u” must be followed by at least four hexadecimal digits.
- © Character codes above 255 are not supported.

## Characters

The OPAL character set comprises all printable Latin-1 (ISO-8859-1) characters as well as space, tabulator, and newline. Other characters can only be used in comments. The character set divides into five classes:

<i>Char</i>	=	$Letgit \cup Special \cup Extra \cup Blank \cup Other$
<i>Letgit</i>	=	{ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z ß à á â ã ä å æ Ç è é ê ë ì í î ï eth ñ ò ó ô õ ö ø ù ú û ü ý thorn ÿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Eth Ñ Ò Ó Ô Õ Ö Ø Ù Ú Û Ü Ý Thorn 0 1 2 3 4 5 6 7 8 9 }
<i>Special</i>	=	{ ! # \$ % & * + - . / : ; < = > ? @ \ ~   ^ _ ‘ } ı cent £ currency yen brokenbar §¨ © ª « ¬ softHyphen registered ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » $\frac{1}{4}$ $\frac{1}{2}$ $\frac{3}{4}$ ¿ × ÷
<i>Extra</i>	=	{ " ( ) , ' [ ] }
<i>White</i>	=	{ space tab nl noBreakspace }
<i>Other</i>	=	all non-printable characters

# Appendix B

## Changes

This chapter lists changes since the 4<sup>th</sup> edition.

- Laws are allowed in signature and implementation parts. (See Section A.3.) Property parts are no longer needed.
- “COMPLETELY” import allowed in signature parts. (See Section A.3.)
- Bracketing rules for infix expressions have been added. (See Section A.6.1.)
- A single dot is recognized as keyword only after one of the keywords `\\ ALL EX`. In particular, “.” is a valid name for a function. (See Section A.9.)
- Escape characters within denotation constants are defined. (See Section A.9.)
- Unicode escapes are recognized. (See Section A.9.)
- The character set has been extended to Latin-1 (ISO-8859-1). (See Section A.9.)

## Appendix C

# Acknowledgement

The language OPAL has been designed by Gottfried Egger, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, Michael Jatzeck, Peter Pepper, and Wolfram Schulte.

Our colleagues and many students have contributed by critically assessing many aspects of the design and implementation, by teaching OPAL, and last not least by giving their invaluable feedback as users of OPAL.

# Bibliography

- [1] Bauer, F.L. et al.: “*The Munich Project CIP*”, Volume I: The Wide Spectrum Language CIP-L; Springer, Lecture Notes in Computer Science 183, 1985
- [2] Bird, R., Wadler, P.: “*Introduction to Functional Programming.*”; Prentice-Hall, 1988
- [3] Diller, A.: “*Compiling Functional Languages.*”; John Wiley, 1988
- [4] Ehrig, H., Mahr, B.: “*Fundamentals of Algebraic Specifications 1.*”; Springer Verlag, 1985
- [5] Field, A.J., Harrison, P.G.: “*Functional Programming.*”; Addison-Wesley, 1988
- [6] Henderson, P.: “*Functional Programming: Application and Implementation.*”; Prentice-Hall, 1980
- [7] Henson, M.C.: “*Elements of Functional Languages.*”; Blackwell Scientific Publications, 1987
- [8] Hudak, P. et al.: “*Haskell.*” Yale University Internal Report, 1990
- [9] Milner, R., Tofte, M., Harper, R.: “*The Definition of Standard ML.*”; MIT Press, 1990
- [10] Perry, N.: “*Hope+. Dept. of Computing*”; Imperial College London Internal Report IC/FPR/LANG /2.5.1/7, 1988
- [11] Perry, N.: “*Hope+C, A Continuation extension for Hope+*”; Dept. of Computing Imperial College London Internal Report IC/FPR/LANG/2.5.1/21, 1987
- [12] Peyton Jones, S.L.: “*The Implementation of Functional Programming Languages*”; Prentice-Hall, 1987
- [13] Peyton Jones, S.L., Wadler, P.: “*Imperative functional programming*”; in: Proceedings of 20th Symposium on Principles of Programming Languages, 1993
- [14] Reade, C.: “*Elements of Functional Programming*”; Addison-Wesley, 1989
- [15] Sanella, D., Tarlecki, A.: “*On Observational Equivalence*”; in: Journal of Comp. and Sys. Science, n° 34, 1987



- [16] Wadler, P.: “*The essence of functional programming*”; in: Proceedings 19th Symposium on Principles of Programming Languages, 1992
- [17] Waite, W. M., Goos, G.: “*Compiler Construction*”; Springer Verlag, 1984