The OPAL Tutorial

The OPAL Group

Jürgen Exner

Technische Universität Berlin
Sekretariat FR 5-13
Franklinstr. 28–29
D–10587 Berlin

jue@cs.tu-berlin.de

May 1994

Report No. 94-9

Abstract

This tutorial describes the functional programming language OPAL which was developed by the OPAL Group at the Technische Universität Berlin.

It explains how to use OPAL from an intuitive approach. No prior knowledge of functional programming will be assumed. Although this tutorial explains OPAL in depth, it does not define the language. In cases of ambiguity, you should consult the reference manual "The Programming Language OPAL".

Contents

Chapter 1

Introduction

A language that does not affect the way you think about programming is not worth knowing.

Author unknown

In the past programming has been dominated by the traditional style of imperative programming. Programming languages like Fortran, Cobol, Algol, Pascal, C and even Assembler are familiar examples of the imperative programming paradigm.

These languages have been oriented towards the internal architecture of the well-known von-Neumann Computer. This implies the main disadvantage of imperative languages: programming must be oriented towards the architecture of the computer instead of towards the structure of the problem to be solved.

As early as 1978 J. Backus asked in his Turing Award Lecture "Can Programming be Liberated from the von Neumann Style?" Since then increasing effort has been invested in the development of alternatives to imperative programming languages. Some of the results are known nowadays by the catchwords 'logic programming' (e.g. Prolog), 'object oriented programming' (e.g. Smalltalk, C++) and 'functional (or applicative) programming' (e.g. pure LISP, ML, HOPE, Haskell).

The programming language OPAL lies somewhere between other modern functional programming languages like ML, HOPE and Miranda. OPAL is a pure functional language without any imperative relics. In addition to higher-order functions, lambda abstraction and pattern-matching, OPAL offers a comfortable modularization ("programming in the large") and a powerful, orthogonal type system which includes generic functions (realized by parameterized structures) and free types. OPAL also supports overloading of names (together with a concise and flexible method for annotation), object declarations, non-deterministic case-distinctions and, in addition, a new way of handling infix- and postfix-operators.

In the past functional programming languages have been accused of inefficiency with respect to time and space. The OPAL research project has defeated

this legend. By using innovative techniques during compilation the runtime of the generated object code is of the same magnitude as for hand-written C-code. This has been proved in several benchmark tests and sometimes, in very special cases, the generated code is even more efficient than a comparable hand-written C-program.

In any case, the generated code is much faster than that of traditional functional languages. There are orders of magnitudes between the execution times of OPAL and e.g. ML or HOPE (see [SchGr92] for details).

This combination of features seems to be unique and we would therefore like to recommend the use of OPAL in research, application programming and education.

1.1 Aim of this Tutorial

In this tutorial we will not assume the reader to be familiar with functional programming or any other programming paradigm. In fact, being familiar with imperative languages , for example, may be disadvantage, because you will have to alter your way of thinking about programming, whereas a novice user is spared this handicap.

Nevertheless, it might be helpful to have some basic knowledge about the theory of programming languages or at least about programming in general.

With a view to accommodating users from all fields the goal of this tutorial will be twofold:

- On the one hand, we will present a short, but complete introduction to the techniques and methods of functional programming.
- On the other hand, we will give a complete introduction to programming with OPAL.

After reading this tutorial a novice user without prior knowledge of functional programming should be able to develop programs in functional style using all the usual features of functional programming, and to implement these programs in OPAL.

But remember, you cannot learn a programming language only by reading about it, you have to write your own programs and learn by use. We therefore advise the reader to try the examples in this tutorial, to modify and enlarge them, and then to write original programs.

1.2 Structure of this Tutorial

Each chapter of this tutorial treats one aspect of OPAL in depth. In general we will use bottom-up methodology, i.e. we will start with the smallest parts of a

program (the names) and finish by combining complete libraries to programming systems.

The advantage of this scheme—the information on each topic will be concentrated in one place and can be easily found by reference to the table of contents—unfortunately also implies a great disadvantage, particular for an introductory tutorial. A novice reader will be overwhelmed by a flood of information he certainly does not need in the initial stages.

We try to avoid this by using a second dimension in the structure. Each paragraph will be marked with a sign that indicates the target group of this paragraph:

$\boxed{\mathcal{N}}$ A section marked with an $\mathcal{N}$ like this one addresses a novice user with no prior knowledge of Opal or functional programming. It contains fundamental information describing the most basic features of Opal, which are essential for trivial programs.

After reading these sections a novice user without prior knowledge should be able to write, compile and execute simple (indeed very simple) Opal programs.

These novice sections are really low-level. They do not explain any of those features which determine the power of functional programming, or advanced features of Opal.

Nevertheless they contain vital information about Opal and should therefore *not* be skipped by experienced users of functional languages.

$\boxed{\mathcal{A}}$ The more advanced features will be explained in sections for advanced users, marked with an $\mathcal{A}$ like this. A novice user should skip these sections on the first run, whereas a user already familiar with functional programming may read them first time round.

The advanced sections contain all the information needed to harness the full power of functional programming and Opal. The concepts and features will be explained in detail, and restrictions and circumventions will be noted. In addition, topics only touched on in the novice sections will be discussed in depth.

In these sections we presume the reader to be familiar with the basic concepts and notations of Opal. Often there will be cross-references to other sections and topics, since, due to the relations between the different language concepts, individual parts of a programming language cannot always be explained in isolation Therefore the reader should be aware that he will sometimes have to read a different section first before being able to understand the current one.

This concept might be considered disadvantageous, but it is the only way to get a concise and also complete reference for a programming language.

$\boxed{\mathcal{E}}$ The third kind of sections are those for experienced users, marked with an $\mathcal{E}$ like this. These sections can be skipped by the normal user altogether, as they are only for the experts. They contain additional background information about special topics and hints for very special features which won't be used by the average application programmer.


Sometimes a paragraph will be preceded by an exclamation mark like this. These paragraphs contain important information and warnings.

1.3 Notational Conventions and Terminology


In order to make this paper easier to read, we will use some conventions for notations and terminology. All these conventions will be detailed a second time as soon as a notion or notation is used in the following chapters. So this section serves mainly as a kind of glossary and may be skipped on a first reading.

Notations


We will use different fonts to distinguish different objects.
The normal font, as used in this chapter, will be used for flow text, explanations, remarks and so on.

Program text will be written in a tty-like font `like this`. We will also use this font in flow text for program fragments (e.g. names of functions), if they belong to a concrete program.

Note that the leading numbers of programs in examples are *not* part of the program. They are only used to reference lines in the explanations.

In interactive examples the output of the computer will likewise be denoted in `tty-like style` while the user's input will also be `underlined`.

When arguing about concepts we won't use the tty-like style, but prefer a more mathematical notation in *a font like this*.

Notions


The notions declaration, definition, signature, implementation and specification are sometimes applied imprecisely in literature. Let us explain their meanings as used in this tutorial by an example:

We want to write a function which doubles its argument. This is already an (informal) specification. A *specification* describes *what* should be done, e.g. doubling the argument. Specifications are essential for arguing about programs, especially for program verification.

The *signature declares* the formal frame:

```
FUN double : nat -> nat
```

The function `double` will take one natural number as argument and deliver a natural number as the result. The signature does not describe what a function does or how this will be done.

The *implementation defines* how the function works:

DEF `double(n) == n + n`

Sometimes we will use declaration and definition as synonyms for signature and implementation.

We won't deal with this topic in detail here, but will return to it repeatedly in following chapters.

1.4 Release Notes

This tutorial describes Opal, Version 2.1, released in Spring 1994. There are some features in this new release, which are not supported by former versions of Opal. These upgrades include:

- sections
- enhanced infix notation
- underscore as wildcard in pattern-matching
- underscore as combinator for alpha-numerical and graphical identifiers
- sequential guards
- compiler now assumes right associate operators if brackets are missing

Consult the appropriate sections if you want to know more about the new features.

The examples in this tutorial are based on "Bibliotheca Opalica" of Spring 1994, as distributed together with the compiler. This library has been considerably restructured and enhanced. See [Di94] for upgrading old programs to the new library.

Chapter 2

A First Example

Before starting with the precise description of OPAL, let us first present a short overview using two introductory examples.

The first is the famous "HelloWorld" program. The second (a little bit more complex and incorporating simple interactive I/O) calculates the rabbit numbers invented by the Italian mathematician Fibbonacci.

These examples are intended only as short survey and so we won't discuss all the details; this will be left to the following chapters.

A third example—not included in this tutorial—can be found in "The Programming Language OPAL " [Pe91, p. 27]. It displays the contents of a named file on the terminal.

A fourth example ("expression") will be included in the appendix. It simulates a small pocket calculator and illustrates user-defined types, higher-order functions and more complex interactive I/O.

Note: All these examples are included in the OPAL distribution and can be found in the subdirectory `examples`.

2.1 "Hello World"

In the world of programming, writing the words "Hello World" on the terminal seems to be absolutely imperative. An OPAL program which does this would probably look like Figures 2.1 and 2.2.

In this example the program consists of one structure named `HelloWorld`, which is stored physically in two files (`HelloWorld.sign` and `HelloWorld.impl`). The files have to be named using the name of the structure plus the extension `.sign` or `.impl`. So the possible names for structures are restricted due to the naming conventions of the file system used.

The signature part declares the export interface of a structure. In the case of a program this must be a constant (e.g., a function without arguments) of

```
1    SIGNATURE HelloWorld
2    IMPORT  Void            ONLY void
3            Com[void]       ONLY com
4    FUN hello : com[void]    -- top level command
```

Figure 2.1: HelloWorld.sign

```
1    IMPLEMENTATION HelloWorld
2    IMPORT  DENOTATION      ONLY denotation
3            Char            ONLY char newline
4            Denotation      ONLY ++ %
5            Stream          ONLY output stdOut
6                      write : output ** denotation -> com[void]
7
8    -- FUN hello:com[void] (already declared in Signature-Part)
9    DEF hello ==
10        write(stdOut, "Hello World" ++ (%(newline)) )
```

Figure 2.2: HelloWorld.impl

sort `com[void]`[1] whereby the sorts `com` and `void` must be imported from their corresponding structures `Com` and `Void`.

In the implementation part we need some additional sorts (`denotation`, `output` and `char`) and operations (`stdOut`, `write`, `%`,`++` and `newline`) which are also imported from their corresponding structures.

Line 8 is a comment line, indicated by a leading "`--`".

The definition of the constant `hello`, which was declared in the signature part, defines this function to write a text to `stdOut` (which is a predefined constant describing the terminal).

The text consists of the words "Hello World" and a trailing newline character, which is converted into a denotation with the operation `%` and appended to the text by the function `++`. The function `++` is used as an infix operator in this example, but this is not essential.


To compile the program you should ensure that the Opal Compilation System (OCS) is properly installed at your site and that the OCS directory `bin` is included in your search path. The GNU `gmake` must be available too. If you don't

[1]The type system will be explained in Chapter 6, instantiations ([...]) in Chapter 7 and the I/O-system in Chapter 5

know how to set up your path or if OCS is not installed, call a local guru.

Within the proper environment—assuming the program `HelloWorld` resides in the current working directory—you just have to type

```
> ocs -top HelloWorld hello
```

to compile and link the program `HelloWorld` with top-level-command `hello` and you will receive an executable binary named `hello`. You can start this program just by typing

```
> ./hello
```

For more information about using OCS try

```
> ocs help
```

or

```
> ocs info
```

and consult the OCS-guide "A User's Guide to the Opal Compilation System" [Ma93] and the man-pages.

A pseudo-interpreter (oi) for Opal programs is also available. Although this interpreter is not intended for complete programs, it is very helpful in the development of separate structures as it simplifies testing considerably. For details, see "The Opal Interpreter" [Le94].

2.2 Rabbit Numbers

Let us have a second example. Imagine a population of rabbits which propagate according to the following rules:

- In the first generation[2] there is only one young couple of rabbits.
- In each following generation the former young couples become grown-ups.
- In each generation each already grown-up couple produces one couple of young rabbits.
- Rabbits never die.

To calculate the total number of couples you may combine the two functions and you will receive the so-called Fibbonacci-Numbers. We will just call them *rabbits*:

$$rabbits(gen) = \begin{cases} 1 & \text{if } gen = 0 \\ 1 & \text{if } gen = 1 \\ rabbits(gen-1) + rabbits(gen-2) & \text{if } gen > 1 \end{cases}$$

```
1    SIGNATURE Rabbits
2
3    IMPORT  Void            ONLY void
4            Com[void]       ONLY com
5
6    FUN main : com[void]    -- top level command
```

Figure 2.3: Rabbits.sign

In the example (Figure 2.4) this formula can be found in the definition of the function `rabbits` (lines 35–39), which is a direct 1-to-1-translation of the mathematical notation.


This direct translation is typical for functional programming. In contrast to traditional imperative languages, you don't need to think about variables and their actual values (which must be supervised very carefully), about call-by-value or call-by-reference parameters or about pointers to results and dereferencing them. This also applies to real problems, not only to such trivial examples as the rabbit numbers.


This example also illustrates local declarations as another feature of Opal. In lines 24–30 three local declarations are established as notational abbreviations:

- `generation` stands for the number the user has typed,
- `bunnies` is the computed number of couples,
- and `result` is the final answer (as a text) of the program.

By using these abbreviations the logical structure of the program will be emphasized and the main action can be notated in a very short form:

```
write(stdOut, result)
```

Local declarations will be detailed in Chapter 4, Section 4.3.7

[2] As we are good computer scientists we will start numbering at 0.

```
1    IMPLEMENTATION Rabbits
2
3    IMPORT  Denotation      ONLY  ++
4            Nat             ONLY nat ! 0 1 2 - + > =
5            NatConv         ONLY ‘
6            String          ONLY string
7            StringConv      ONLY ‘
8            Com             ONLY ans:SORT
9            ComCompose      COMPLETELY
10           Stream          ONLY input stdIn readLine
11                              output stdOut writeLine
12                              write:output**denotation->com[void]
13
14   -- FUN main : com[void] -- already declared in signature part
15   DEF main ==
16       write(stdOut,
17          "For which generation do you want
                    to know the number of rabbits? ") &
18       (readLine(stdIn)    & (\\ in.
19        processInput(in‘)
20       ))
21
22   FUN processInput: denotation -> com[void]
23   DEF processInput(ans) ==
24             LET generation == !(ans)
25                 bunnys     == rabbits(generation)
26                 result     == "In the "
27                               ++ (generation‘)
28                               ++ ". generation there are "
29                               ++ (bunnys‘)
30                               ++ " couples of rabbits."
31             IN writeLine(stdOut, result)
32   -- ------------------------------------------------------
33
34   FUN rabbits : nat -> nat
35   DEF rabbits(generation) ==
36           IF generation = 0 THEN 1
37           IF generation = 1 THEN 1
38           IF generation > 1 THEN rabbits(generation - 1)
39                                  + rabbits(generation - 2) FI
```

Figure 2.4: Rabbits.impl

Chapter 3

Names in Opal

Names are the basis for all programming because you need names to identify the objects of your algorithm. In Opal the rules for constructing names are more complex than in most traditional languages for two reasons:

First, Opal also allows the construction of identifiers with graphical symbols like "`+`", "`-`", "`%`" or "`#`", which can be used the same way as the established identifiers made up of letters and digits. This will be explained in the following section.

Second, Opal supports overloading and parameterization and thus requires a method for annotations (see Section 3.2: "What's the name of the game?").

3.1 Constructing Identifiers

In this section we will explain which characters can be used to build an identifier and which rules have to be fulfilled.

For constructing an identifier all printable characters are divided into three classes:

- upper-case letters, lower-case letters and digits (e.g., "`A`", "`h`", "`1`")
- graphical symbols: these are all printable character with the exception of letters, digits and separators[1]. Examples are "`+`", "`$`", "`@`", "`{`", "`!`"
- separators: these are "`(`", "`)`", "`,`", "`'`", "`"`", "`[`", "`]`" together with space, tabulator and newline. The last three are often called "white space".

You may construct identifiers from either of the first two classes. This means "`HelloWorld`", "`a1very2long3and4silly5identifier6with7a8lot9of0digits`", "`A`", "`z`", "`2345`", "`+`", "`#`", "`---->`", "`%!`" are legal identifiers in Opal.

[1] Although question mark "`?`" and underscore "`_`" belong to this group too, they have special meanings (see below).

But note that you cannot mix letters and digits with graphical characters in one identifier, e.g., "`my1Value<@>37arguments`" is a list of three identifiers "`my1Value`", "`<@>`" and "`37arguments`".

The case of a letter is significant, so "`helloworld`", "`HelloWorld`" and "`HELLOWORLD`" are three different identifiers.

Separators cannot be used in identifiers at all. They are reserved for special purposes and delimit any identifier they are connected to.

Summarizing, one can say an identifier is the longest possible sequence of characters, either of letters and digits or graphical characters.

3.1.1 Question Mark and Underscore


Although question marks are graphical symbols, they can also be used as trailing characters of an identifier based on letters or digits. This exception was introduced because the discriminators of data types are constructed by appending a question mark to the constructor (for details, see Chapter 6: "Types").


An underscore-character "_" has a special meaning too. First it is a member of both character classes, letters as well as graphical symbols. Hence it can be used to switch between the character classes within one identifier. E.g., "`my1Value_<@>_37arguments`" is—in contrast to above—only one identifier.

Furthermore, a single underscore is a reserved keyword with two applications (as wildcard and as keyword for sections, see 6.3.1 and 4.3.6 for details). Therefore you should be careful when using underscores.

3.1.2 Keywords


Some identifiers are reserved as keywords, e.g., "`IF`", "`IMPLEMENTATION`", "`FUN`", "`:`","`_`", "`->`" (see "The Programming Language Opal" for a complete list of all keywords).

These keywords cannot be used as identifiers any more, but it is no problem to use them as part of an identifier: "`myFUN`", "`THENPART`","`IFthereishope`","`::`" and "`-->`" are legal identifiers.

Moreover, because upper and lower case are significant, "`if`" and "`fun`" are legal identifiers and not keywords.


This is a common reason for curious errors. A programmer will write

```
FUN help : ...
```

to start the declaration of the function `help`. The line

```
FUNhelp : ...
```

obviously means something completely different (in this case it will be an error), but a programmer will recognize this error at once.

The same error—a missing space—written with graphical symbols is much less obvious! You must take care to write

```
FUN # : nat -> nat
```

instead of

```
FUN #: nat -> nat
```

So, if you receive a curious error message, first check that you have included all necessary separators between identifiers and keywords.

3.2 "What's the name of the game?"

In the previous section we discussed the construction of single identifiers. For several reasons—the two most important are overloading[2] and parameterized structures (see Section 7.3)—an identifier alone does not suffice to really identify an object under all circumstances. It is for this reason that an identifier can be annotated with additional information.

By carefully analyzing the environment of an identifier, the compiler will nearly always detect by itself which operation to be used. In very complex cases this detection may fail and you will receive an error message saying something like "ambiguous identification". In these cases you can annotate the identifier to help the compiler.

In addition, you must annotate the instance of a parameterized structure at least once, either at the import or at the application point (see Chapter 7.3 for details).

Annotations are always appended to an identifier, a white space[3] may be added between identifier and annotations, but this is not necessary.

The following annotations in particular are possible. You may omit each of them, but if you supply two or more, they must be supplied in the order presented below:

- **Origin:** The origin of an object is the name of the structure in which this object was declared.

  The annotation of the origin is introduced by a "'", followed by the name of the origin structure.

[2]Using the same identifier for different functions is called overloading. In traditional programming languages this feature is common for built-in data types (e.g., the symbol + for addition of natural and real numbers), but it's very rarely supported for user-defined functions.

[3]Blanks, tabulators and newlines

E.g., `-'Nat` identifies the function `-` from the structure `Nat`, whereas `-'Int` identifies the function `-` from the structure `Int`.

- **Instantiation:** The instantiation of an object of a parameterized structure (see 7.3 for details) will be annotated by appending the parameters to the identifier in square brackets. E.g., if you want to use the function `in` from the library structure `Set` with a set of natural numbers you may annotate `in[nat,<]`—or in even more detail— `in'Set[nat'Nat, <'Nat]`.

- **Kind:** The kind of an object is either the keyword "`SORT`" or the functionality of the object. The kind is appended with a "`:`". E.g., the name `- :int->int` identifies the unary minus, whereas `- :int**int->int` identifies the usual dyadic minus[4].

[4]Remember the space between `-` and `:`.

Chapter 4

Programming in the Small

In functional languages the algorithms are expressed in terms of functions—as the name "functional programming" already implies. In this chapter we will explain how to declare and define functions (Section 4.1) and how to use functions to express algorithms (Section 4.3). We will also describe the rules for scoping and visibility of names (Section 4.2).

This chapter is called "Programming in the Small" because functions are used to structure a program in a fine grain. There are also features for structuring a program in a coarse grain, which means summarizing several functions (and data types) in structures of their own. This will be explained in Chapter 7: "Programming in the Large".

The examples in this chapter are generally taken from the two example programs, `HelloWorld` and `Rabbits` (see Chapter 2: "A first Example"), for the novice, and from `Expressions` (see Appendix B.2) for the more advanced features. Before reading the advanced paragraphs you should first read about data types (see Chapter 6: "Types"), because the program "`Expression`" uses a lot of data type definitions and the algorithms are based on these.

4.1 Declaration and Definition of Functions

Functions are the basis of functional programming and they are the only way to express algorithms in purely functional languages like Opal. Although traditional imperative languages generally also offer functions, their usage is restricted by several constraints.

In Opal functions are much more general and there is in fact no difference between functions and ordinary values as is the case in imperative languages. Both can be used in exactly the same way (this is sometimes apostrophized as "functions as first-class-citizens"). From now on we will just say "object" if we don't want to distinguish between functions and ordinary elements of data types.

To write a function you perform two steps: you have to declare and you have to define the function. Both steps will be explained in the following.

4.1.1 Declaration of Functions


The declaration tells about the *kind* of arguments and results of a function. It does not fix what the function will do (this is the task of the specification) or how this will be done (this is the object of the definition; see below).

A function declaration is introduced by the keyword `FUN`, followed by the name of the function, a colon and the functionality of the function. The declaration

```
FUN rabbits : nat -> nat
```

declares the function named `rabbits`, which will take one natural number as argument and deliver a natural number as result.

The argument (and also the result, see below) could be tuples:

```
FUN add : nat ** nat -> nat
```

This means the function `add` will take two natural numbers as arguments and deliver one natural number as result.

Remember, a declaration does not say anything about what the function will do. The function `add` might deliver as result the minimum of the two arguments; this would be rather contra-intuitive and should therefore be avoided.

The arguments may be missing altogether as in

```
FUN main : com
```

This means the function `main` takes no argument, and in this case we say `main` is a constant of the sort `com` (which means `command`; see Chapter 5: "Input/Output in OPAL" for details about commands).

The sorts of arguments and results need not to be the same. The function

```
FUN ! : string ** nat -> char
```

as declared in the library structure `StringIndex` takes as arguments a string and a natural number and delivers a character.

Remember that graphical symbols are allowed as identifiers and that the space between "`!`" and "`:`" is very important (see Section 3.1 if you'd forgotten about that).

As you could see above, the functionality is always expressed in terms of sorts. These sorts must be known in the structure, i.e. they must be either imported or declared in the structure.


Arguments and results of a function could be much more general than explained above.

Thus the result of a function could also be a tuple. This might be useful, e.g., in a function

```
FUN divmod : nat ** nat -> nat ** nat
```

which returns the quotient and the remainder of a division simultaneously (on how to select the elements of a tupled result, see Section 4.3.7: "Object Declarations").


Theoretically, the number of arguments is unlimited; in the current implementation it is restricted to 16. This is not a serious restriction, because 16 is quite a large amount. IF you really do need more parameters, you can combine several arguments into a new data type to reduce the number of arguments.

Higher-Order Functions


You can have functions as arguments and results too. These functions are called higher-order functions. Usually higher-order functions are supported only very rudimentarily in traditional languages, if at all.

A function to compute the integral may be declared like this:

```
FUN integral : (real -> real) ** real ** real -> real
```

which means that the first argument is the function to be integrated and the second and third argument are real numbers defining the lower and upper bound of the integral.

Another example can be found in the program `Expression`:

```
FUN doDyop : dyadicOp -> nat ** nat -> nat
```

The function `doDyop` takes an element of the sort `dyadicOp` and delivers a function which itself takes two numbers as arguments and delivers one number as result.

The symbol `->` is right-associative. This means the declaration above is equivalent to

```
FUN doDyop : dyadicOp -> (nat ** nat -> nat)
```

and the parentheses may be omitted.

This process of functions as arguments or results could be continued. There is no limit to the "nesting depth".

Currying


As known from theoretical computer science and mathematics, it is always possible to transform a function which takes more than one argument into a function which takes only one argument and delivers a function as result, without altering the semantics of the function. This process is called *currying*.

```
FUN + : nat ** nat -> nat
FUN + : nat -> nat -> nat
```

The second variant is the curried version of the first.

In some functional languages this transformation is done automatically and the two declarations are recognized as identical.

In OPAL the two declarations are distinguished and need their own definitions. Each variant has its own advantages: with the first the function `+` could be used as infix-operator, whereas with the second one you can define a function, e.g. `+(3)`, a brand new function which will add three to its only remaining parameter. This can be useful, for example in conjunction with other higher-order functions, such as apply-to-all on sequences.

On the other hand, if you have used the uncurried version, you can use sections and lambda expressions (see 4.3.6 and 4.3.5) to do a "currying on the fly", i.e. to define a temporary, auxiliary function with partially supplied parameters. It is therefore simply a matter of taste, whether you prefer the curried or the uncurried version.

4.1.2 Definition of Functions


The definition of a function defines how the function works, i.e. it represents the real algorithm.

The definition of a function consists of a header on the left side of the "`==`" and a body on the right side. The function `rabbits` will be defined by

```
DEF rabbits (n) ==  << body >>
```

The number of parameters ("`n`") must match the number of parameters given in the declaration of the function (in this example one argument). In the body the name `n` will be visible (see next section). And due to the declaration of `rabbits` it will stand for an object of type `nat`. The parameters are used to reference the arguments of a concrete call of `rabbits` in the body.

The body of a function definition is an expression. We will explain expressions in Section 4.3.

The headers of the other examples from the previous section ("Declaration of Functions") will be something like

```
DEF main ==  << body >>
DEF !(str, n) ==  << body >>
```

They define `main` to have no parameters and `!` to have two parameters, named `str` and `n`, which can be used in the body of the definition.

The functionality[1] of the parameters can be derived from the declaration of the corresponding function. Function "`!`" is declared as `FUN ! :string**nat->char`.

[1] Functionalities are required for checking the correctness of expressions; see Section 4.3 below.

Therefore the first parameter, `str`, has functionality `string` and the second, `n`, has functionality `nat`.


There is a bunch of other ways to notate the header of a function definition. First of all you can use infix-notation which is quite similar to infix expression 4.3.3. You could also write

```
DEF str ! n ==  << body >>
```

instead of the definition above.

Furthermore, you can use pattern-matching to define functions. Pattern matching depends on free types and will be explained in Section 6.3: "Pattern-Matching".

Higher-order functions with functions as arguments are defined the same way as first-order functions. The function `integral` (see above) could be defined as

```
DEF integral (f, low, high) ==  << body >>
```

where `f` denotes the function to be integrated.

The functionality of the parameters is naturally extended: `f` has functionality `real->real`, `low` and `high` have `real` respectively.

Higher-order functions with functions as results will be defined with additional parameters for the parameters of the result function. The function `FUN doDyop :  dyadicOp -> nat ** nat -> nat` could be defined as

```
DEF doDyop (op)(l, r) ==  << body >>
```

In this case `op` is a dyadic operand and `l` and `r` are the natural numbers as arguments for the resulting function.

More concretely, this means you define this function as

```
DEF doDyop (op)(l, r) == IF op addOp? THEN +(l,r)
                         ...
```

The functionality of the parameters again are naturally extended: `op` has functionality `dyadicOp` and `l` and `r` have `nat`.

But you can even shorten this header. If you want to define a function `myadd :  nat ** nat -> nat` with the same semantics as the standard definition of addition (i.e. renaming the function `+`) you can do this by writing:

```
DEF myadd (a,b) == +(a,b)
```

It is not true however, that for each parameter in the declaration there must be a corresponding parameter in the definition. As long as the type remains correct you can omit the parameters.

```
DEF myadd == +
```

On the right side there is only one identifier with the functionality `nat**nat->nat`, and the left side has the same. So this definition is correct.

Omission of parameters can only be done on whole tuples. This means you cannot leave out `b` alone. You must write either all or none of the parameters of a tuple. And—of course—you can only omit trailing tuples.

This scheme is especially useful with higher-order functions. You can shorten the definition of the function `doDyop` to

```
DEF doDyop (op) == IF op addOp? THEN +
                   ...
```

where the function yields—depending on the operation `op`—just one of the well-known functions `+, -, *, /`.


Omitting parameters entails one small catch. In Opal all constants are evaluated only once during the initialization phase of the program. Constants are function *definitions* without parameters. Therefore the second definition of `myadd` (see above) is a constant definition (it does not depend on arguments), whereas the first is a function definition (it has two arguments).

In general this won't make any difference and you can use the two as you please. But you should note, that the definition of constants must be acyclic, i.e. you can't use a constant in its own definition, neither direct nor transitive.

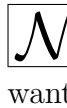
Moreover there are rare cases where the time of evaluation of functions and arguments might be significant. Sometimes it is even useful to add an empty argument tuple to delay the evaluation of a function call until later:

```
FUN f : nat ** nat -> () -> res
```

In this case the function call `LET g == f(1,2) IN ...` won't be evaluated, but yields a closure (i.e. a new function) which could be submitted as an argument or stored in a data structure.

Only if this closure is applied with the empty tuple (e.g. `g()`), then the call `f(1,2)` is evaluated. Using this method you can simulate lazy execution within the strict language Opal.

4.2 Scoping and Overloading


Scoping and overloading generally involve very complex rules. Scoping means: I have declared an object somewhere in the program text and want to know if this object is known (accessible) somewhere else. If the object is accessible, we say it is *visible* at this location.

Overloading means using the same identifier for different objects.

Concerning scoping and overloading in Opal you have to distinguish two different kinds of objects

- global objects
- and local objects.

Global objects are all imported objects as well as all functions and sorts declared in the structure, either in the signature or in the implementation part.

Global objects may have the same identifier as long as they can still be distinguished by annotations (see Section 3.2 for details). This means that if two global objects differ in at least one of their identifiers, their origins, their instantiations or their kinds (sort or functionality) you can use both objects side by side.

If they are the same in identifier and all possible annotations, the compiler will perceive them as identical. This means it is possible, for example, to import the same object several times.

Local objects are parameters, lambda- and let-bound variables (see Section 4.3). They cannot be annotated with origin or instantiation.


The names of local objects must be disjoint within their visibility region, i.e. they cannot be overloaded at all.

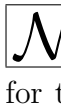

The rules for scoping (i.e. visibility) are fairly simple in Opal. A signature part cannot have local objects. In the signature part the only visible objects are those which are imported or declared in the signature part.

Throughout the implementation part, all global objects from the signature part and the implementation part are visible.

Parameters of function definitions are visible throughout the function definition; lambda- and let-bound variables are visible only within their expressions.


If a local and a global object have the same identifier, the global object will be invisible as long as the local object is visible, *even if the identifier is annotated*. In this case the annotation on local objects is ignored! This does not apply to sort-names, because these can be identified by their position in the program text.

4.3 Expressions


In this section we will explain how to construct the body of a function definition. First we introduce the fundamental expressions essential even for trivial programs; these are atomic expressions, tupling of expressions, function applications and case distinctions. Then we will discuss the more elaborate features which improve the power of functional programming and the readability of Opal programs: lambda abstractions, sections and local declarations.

A basic issue regarding correctness of expressions is the functionality of an expression. The main context condition for expressions demands that functionalities must fit together. Therefore we will also discuss the functionality of each expression and the conditions it must satisfy.

4.3.1 Atomic Expressions

The most simple expressions are atomic expressions. There are two kinds of atomic expressions: identifiers and denotations.

Identifiers denote objects (i.e. functions, local objects and elements of data types) which have to be visible[2] when the identifier is used.

Examples are `rabbits`, `0`, `=`, `+` and `generation` (all are taken from the example program `rabbits`). In the case of a global identifier the functionality is declared in the identifiers declaration; the functionality of a local identifier can be derived either from the declaration of the corresponding function (for parameters; see Section 4.1.2) or from the context (for local declarations and lambda-bound variables; see Sections 4.3.7 and 4.3.5 for details).

Denotations are special notations for denoting arbitrary objects. They are enclosed in quotation marks and are often used to represent text, e.g. "`Hello World`", but you can write conversion routines to represent just about every data object using denotations. The corresponding conversion functions for natural numbers, integers and reals are predefined in the standard library.

These conversion functions are usually named "`!`". For example `"1234"!'Nat` represents the natural number 1234 (remember, `'Nat` is an annotation defining the origin of "`!`" being the structure `Nat`). More about denotations can be found in Section A.1.

Denotations always have functionality "`denotation`".

4.3.2 Tuples

Expressions (any expression, not only atomic ones) can be grouped together as tuples by enclosing them in parentheses:

```
("Hello World", 1, +(5,3))
```

is a tuple with three elements, a denotation, a number and a function application. Tuples are used mainly to unite the arguments of a function application, e.g. in `+(3, 5)`.

Tuples are always flat in Opal. That is, you can write a nested tuple like `(a, b, (c, d), e)` which seems to consist of four elements, the third being a tuple itself. But in Opal this is identical with the tuple `(a, b, c, d, e)`.

A tuple may contain only one element while the empty tuple is allowed only as an argument of function calls.

Flat tuples allow some compact notations on using functions which return tuples as results. Suppose you have a function `f:nat**nat**nat->nat` which

[2] see explanation of visibility above.

takes three arguments. Remember the function `divmod:nat**nat->nat**nat` which returns a tuple of two numbers.

You can write `f(3, divmod(22,5))` which is the same as `f(3, (4, 2))` (the function call `divmod(22,5)` being evaluated and—because tuples are flat—this is a correct call of `f(3, 4, 2)`).

4.3.3 Function Applications


A function application consists of two expressions, the second being a tuple, e.g. `+(3,4)` or `rabbits(generation-1)`. The tuple consists of the arguments of the functions call.

The functionality of the first expression must be a function functionality consisting of parameters and a result. The functionality of the tuple must match the functionality parameters. The functionality of the whole function application is the result.

In the example above the first expression is "`+`", which has a functionality of `nat**nat->nat`. The second expression is the tuple "`(3,4)`" with functionality `nat**nat`, which matches the parameters of "`+`". The functionality of the application is `nat`, the result part of the functions functionality.


There are several variations on the notation of function applications. Each function application could also be written as a postfix-operation, i.e. just exchange the two expressions: `(3,4)+`.

In this case the parentheses around a tuple containing only one element may be omitted. Postfix-operations are often used to shorten the notation for conversions like `"Hello World"!` which is the same as `!("Hello World")`.

Postfix-operations are also known from mathematics;for example the factorial is usually written as 3!.

Function applications may also be written as infix-operations: `(3)+(4)`. Again you can omit the parentheses if the tuples only contain one element. This results in the usual notation for mathematical expressions: `3+4`.

In general graphical symbols like `+`, `*`, `!`, `::` or `+%` are used as function names if the function is intended to be used as an infix- or postfix-operation, but this is not necessary.

The definition and the application of a function are independent. You can define a function using pattern-based definitions and infix-notation and you can still apply the function in traditional prefix style.

Infix-notation could also be used with more than two arguments:

```
f(a,b,c); f(a, (b,c)); a f (b,c); (a) f (b,c); (a,b) f c; (a,b,c)f
```

are all the same function application.

You could also nest infix-notation, but there is one little problem with this very flexible notation. Because the analysis of arbitrary infix notations is very expensive, you should use at most 5–6 infix operations without bracketing. If your expression is larger, you should enclose subexpressions in parentheses. This will also enhance readability.

This problem seems ridiculous, especially as even traditional imperative languages can deal with any number of infix-operations. But remember a "`+`" in OPAL could be any function, including a user-defined function or a function computed at run-time (in contrast to traditional languages), and this much more general problem has not been resolved satisfactorily to date.


There is one more problem with arbitrary infix-notation. As the compiler cannot have any knowledge about precedence rules between the different functions (in contrast to C, for example), it can use only the various functionalities to detect which applications were intended by the programmer.

A typical example is the construction of a list (using `FUN :: :data**seq->seq`):

```
1 :: 2 :: 3 :: 4 :: <>
```

Due to the functionalities, the compiler will recognize this as:

```
1 :: (2 :: (3 :: (4 :: <> )))
```

If the compiler cannot resolve this problem with a unique solution, you will receive an error message like `ambiguous infix application` and you should insert some parentheses to help the compiler.


If you are using the same function several times in nested infix-notation, the compiler will assume right-associativity.
The concatenation of several denotations

```
"ab" ++ "cd" ++ "ef" ++ "gh"
```

will be recognized as

```
"ab" ++ ("cd" ++ ("ef" ++ "gh"))
```

WARNING: This is contra-intuitive with respect to normal mathematics! The expression `20-10-5` yields 15 instead of 5 as expected because the compiler assumes right-associative functions and "`-`" is not right associative! In this case you must supply parenthesis: `(20-10)-5`.


The application of higher-order functions is just the same as for first-order functions. Remember the functions

```
FUN integral:(real->real) ** real ** real -> real
```

and

```
FUN doDyop: dyadicOp -> nat ** nat -> nat
```

`integral` will be applied as

```
integral(sin, 0, 1)
```

for example to compute the integral over the sinus function between 0 and 1. Remember, you also can write this application as infix or postfix, e.g.

```
sin integral (1, 2)
```

The function `doDyop` expects one argument of functionality `dyadicOp`. A correct application will be

```
doDyop(addOp)
```

You could also write it as postfix[3]:

```
addOp doDyop
```

The result of this application is a function with functionality `nat**nat->nat`. Thus you can apply this function to two numbers:

```
doDyop(addOp)(3,4)
```

Function application associates to the left, i.e. the expression above is the same as

```
(doDyop(addOp))(3,4)
```

This application could also be written as

```
(addOp doDyop)(3,4)
```

for example.

But note that a function used as an infix-operator must be an identifier. Therefore you cannot write

```
3 (addOp doDyop) 4     -- this is illegal
```

because `(addOp doDyop)` is no identifier.

This example also illustrates another feature. The first expression (i.e., the function of the function application) need not be a name. It can be any expression including another function application (as above), a case distinction, a lambda abstraction or an object declaration. The only context condition is that it must have a functions functionality.

[3]This time you cannot use infix-notation, because there is only one argument.

4.3.4 Case Distinctions


Case distinctions are used to control the evaluation of the program. Depending on the value of conditions, the program will continue to evaluate different parts.

A case distinction in OPAL is written as

```
IF cond_1 THEN part_1
IF cond_2 THEN part_2
...
IF cond_n THEN part_n
FI
```

`cond_1` ... `cond_n` and `part_1` ... `part_n` are arbitrary expressions including function applications, other case distinctions or local declarations.

`cond_1` ... `cond_n` are called guards and all of them must have functionality `bool`. `part_1` ... `part_n` must have identical functionalities and this common functionality is also the functionality of the case distinction.

Let us have a concrete example. The function `rabbits` is defined as:

```
DEF rabbits(generation) ==
    IF generation = 0 THEN 1
    IF generation = 1 THEN 1
    IF generation > 1 THEN rabbits(generation - 1)
                           + rabbits(generation - 2)
    FI
```

The guards are infix function applications, each of functionality `bool`. Each of the THEN-parts has functionality `nat` which is the functionality of the case distinction and also of the result of the function.

The semantics of a case distinction is: evaluate one of the guards. If the guard yields false, forget about this case and try another one. As soon as a guard yields true, stop searching and evaluate the corresponding THEN-part.

If none of the guards yields true, the value of the case distinction is undefined and the program will terminate with an error message.


The order the guards are evaluated in is selected by the compiler! It does not depend on the order in which the guards are denoted in the program!


The example above may be evaluated as follows (supposing `n` has the value `0`): first the program might check the last guard. `0>1` yields `false`, therefore the next guard will be checked; let us now assume it to be the first guard. `0=0` yields `true`. The corresponding THEN-part will therefore be evaluated, yielding `1` and the value of the case distinction is computed as `1`.

If the guards are not disjoint there is no specification as to which THEN-part will be evaluated. Sometimes this is desirable, as in

```
DEF maximum(n, m) ==
          IF n <= m THEN m
          IF m <= n THEN n FI
```

In this case the program will always check only one guard if the values of `n` and `m` are the same.

But you should ensure that the result is the same in any possible case. Otherwise your program might behave unexpectedly.

Remember that OPAL is a strict language and the guards are just expressions. If you have a condition consisting of two parts and the second is valid only if the first yields `true`, you cannot combine the two parts with an `and`-function!

There is a well-known example for this fault: given a list of numbers, you want to compare the first element of the list with some value. The first part of the condition checks if the list is not empty (because otherwise you can't access the first element) and the second part does the comparison.

It is wrong to write

```
IF (~(list empty?) and (ft(list) = 0)) THEN ... -- this is wrong
...
FI
```

because *both* arguments of the `and` will always be evaluated. This results in a runtime-error if the list is empty.

Instead you have to split the condition into two guards in a nested case distinction:

```
IF ~(list empty?) THEN
            IF  ft(list) = 0 THEN ...
            ...
            FI
...
FI
```

This could be abbreviated to the short-hand notation

```
IF (~(list empty?) ANDIF (ft(list) = 0)) THEN ...-- this is correct
...
FI
```

There is also a corresponding `ORIF`. Both operations evaluate their first argument. Only if this evaluation yields `true` for `ANDIF` or `false` for `ORIF`, the second argument will be evaluated too.

There are two notational variations for case distinctions. Firstly, you can add an "`OTHERWISE`" between any two cases. Then the program will first check all guards in front of the `OTHERWISE`, and only if all of these guards yield `false` will it continue with the guards following the `OTHERWISE`.

The expression

```
IF cond_1 THEN part_1
...
IF cond_n THEN part_n
OTHERWISE
IF cond_n+1 THEN part_n+1
...
IF cond_n+m THEN part_n+m
FI
```

is exactly the same as

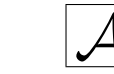
```
IF cond_1 THEN part_1
...
IF cond_n THEN part_n
IF ~(cond_1 or ... or cond_n)
   THEN
      IF cond_n+1 THEN part_n+1
      ...
      IF cond_n+m THEN part_n+m
      FI
FI
```

Secondly, you can add an "`ELSE expr`" after the last case of a case distinction. If the evaluation of all guards yields `false`, the ELSE-expression will be evaluated instead of the program terminating with an error.
The case distinction

```
IF cond_1 THEN part_1
...
IF cond_n THEN part_n
ELSE expr
FI
```

is the equivalent of

```
IF cond_1 THEN part_1
...
IF cond_n THEN part_n
OTHERWISE
IF true THEN expr
FI
```


In addition to the fundamental expressions described above, Opal offers three more constructs for defining expressions. These are lambda abstractions, sections and local declarations. Lambda abstractions and local declarations also enlarge the number of local objects by new objects which are visible inside their expressions (see below).

4.3.5 Lambda Abstraction


Lambda abstractions define auxiliary functions without a name. The notation is

```
\\x,y. expr
```

This lambda expression defines a function which takes two arguments. The parameters are called `x` and `y`. These new local objects are visible only inside `expr`. The functionality of `x` and `y` could either be annotated or it will be derived automatically from their usage in `expr`.
Let us have a concrete example:

```
\\x.x=3
```

defines a function with functionality `nat -> bool`, which compares a number to 3.

You can apply this function: for example `(\\x.x=3)(4)` yields `false`. Or you can use it in an object declaration:

```
LET equalThree == \\x.x=3
IN ...
```

This way you receive a named auxiliary function, although lambda abstractions cannot be recursive.

Very often lambda abstractions are used to submit an auxiliary function to higher-order functions as arguments, e.g.

```
LET a == pi
    b == e
    c == "-5.0"!
IN integral(\\x.a*x*x + b*x + c, 0, 1)
```

computes the integral $\int_0^1 ax^2 + bx + c$ with $a = \pi$, $b = e$ and $c = -5$.

Lambda abstractions may also be nested to define higher-order functions and they can be used to define ordinary named functions: The definitions

```
DEF f(a,b)(c) == expr
```

and

```
DEF f == \\a,b. \\c. expr
```

are equivalent and each of them defines a function with functionality,
e.g. `FUN f:s1**s2->s3->s4`.

As in pattern-matching (see Section 6.3), an underscore character may be used as a wildcard in a lambda abstraction; the meaning is that there is a parameter for a (lambda-defined) function, but I am not interested in its value.

4.3.6 Sections


Sections are a shorthand notation for simple lambda abstractions. You will often need a function where some arguments should not yet be fixed.

As an example you may think of a function that adds the value of 3 to each element of a sequence of numbers `s`. Using the apply-to-all-function "`*`", this could be written with lambda abstraction as

```
* (\\ x. 3+x)(s)
```

or in shorthand form with section as

```
* (3 + _) (s)
```

Note: don't forget the separator between "`+`" and "`_`". Otherwise you apply the (probably undefined) function "`+_`" on the argument 3.

Underscores represent arguments of a function call which are missing at the moment, but will be supplied later.

Sections could be regarded as a generalization of currying, because not only trailing (as with currying) but also arbitrary arguments can be postponed until later.

In detail, the expressions

```
f(a,b,c,d,e);    f(a,b,_,d,_)(c,e);    (f(a,_,_,d,_))(b,_,e)(c)
```

are all the same.

4.3.7 Local Declarations


Local declarations can be used to structure the definition of a single function and to introduce abbreviations.

Local declarations are written as

```
LET o_1 == expr_1
    o_2 == expr_2
    ...
    o_n == expr_n
IN expr
```

or

```
expr WHERE o_1 == expr_1
           o_2 == expr_2
           ...
           o_n == expr_n
```

The two notations are equivalent.

The additional local objects `o_1` ... `o_n` are visible in the expression `expr` and in all expressions `expr_1` ... `expr_n`.

A declaration `o_i == expr_i` is said to depend on the declaration `o_j == expr_j` if `o_j` is used in `expr_i`. The ordering of the declarations is irrelevant, but there must not be cyclic dependencies between the declarations.

The expression `expr` will be as long as possible; in the local declaration

```
LET a == ...
IN f(a)(e_2)
```

for example, the `a` will be visible in the whole expression `f(a)(e_2)`, not only in the expression `f`.

The semantics of local declarations could be explained with lambda abstraction. Assuming the declaration `o_1 == expr_1` does not depend on any of `o_2` to `o_n`, then the local declaration

```
LET o_1 == expr_1
    o_2 == expr_2
    ...
    o_n == expr_n
IN expr
```

is equivalent to

```
(\\ o_1. LET o_2 == expr_2
         ...
         o_n == expr_n
         IN expr           ) (expr_1)
```

The declaration that does not depend on any other must not necessarily be the first one, because ordering of declarations is irrelevant. But because the dependency between the declarations must be acyclic, there is always a declaration, that does not depend on any other.

Note that local declarations are strict. Therefore a declaration like

```
LET cond     == ...           -- dangerous pgm style
    thenpart == ...
    elsepart == ...
IN IF cond THEN thenpart ELSE elsepart FI
```

will not behave as expected, because `thenpart` and `elsepart` are *always* evaluated due to the strict semantics of local declarations.

Chapter 5

Input/Output in OPAL

The problem of interactive input/output (I/O) in functional languages is still a research topic. OPAL uses continuations to support I/O. We won't discuss the theory of continuations here as we prefer to take a more intuitive approach.

The fundamental issue of I/O in OPAL is the *command*. A command is a data object that describes an interaction with the environment, e.g. the terminal or the file system. Only commands do this. The command itself is executed by the runtime-system, which evaluates the commands at runtime according to the rules described in this chapter.

Commands can be determinated by their functionality: a command is always a constant of type `com'Com`. Functions which yield an object of type `com‘Com` as a result are used very frequently. They can be regarded as parameterized commands.

5.1 Output

We have already seen some examples of commands. In the example program, `HelloWorld` (see Chapter 2), the function `write` is a predefined function from the library structure `Stream` with functionality

```
FUN write : output ** denotation -> com
```

The whole program itself interacts with the environment too. Therefore it is also a command (the so-called top-level command which is required for every program):

```
FUN hello : com
```

Commands can be combined to form new commands with the functions "`;`" and "`&`" from the library structure `ComCompose`. These functions are usually writ-

ten as infix operations. Both combination functions take two commands as arguments and combine them to a new one by first executing the first command and afterwards the second. To combine some write-commands you could write

```
writeLine(stdOut, "This will be the first line") &
writeLine(stdOut, "Some more output") &
writeLine(stdOut, "The last line")
```

which will print the three lines to standard output, i.e. usually the terminal.

We have already used command combination in the `Rabbits` example (see Chapter 2):

```
write(stdOut,
        "For which generation do you want to know ...? "!)&
(readLine(stdIn) &
processInput)
```

Since commands describe interactions with the environment there is no certainty that the evaluation of a command will always succeed (for example a requested file can't be accessed or the connection to an internet socket has been broken). Therefore there must be some error-handling.

The functions "`;`" and "`&`" differ in their handling of errors. If an error occurs during the execution of the first command, "`&`" does not execute the second command, whereas "`;`" will. In this case the programmer must check within the second command whether an error has occurred during the execution of the first and continue with an appropriate alternative (e.g. asking the user for another filename if a file could not be opened). How to access and analyze a possible error message will be explained later on.

Note that "`;`" and "`&`" are strict. The second argument will always be computed, but the resulting command will be executed only if execution of the first succeeds. If the execution of the first command fails, the corresponding error is yielded.

5.2 Input

Up to now we have only dealt with commands which produced some output and combinations of them. But for real computations we will need some input too.

For this reason commands are parameterized (see Section 7.3) and can be instantiated with the sort they should yield as result of an input operation. The command `readLine'Stream(stdIn)`, for instance, reads a string from the terminal and delivers this string as result:

```
FUN readLine : input -> com[string]
```

The result cannot be accessed directly, as `readLine` yields a command, not a string. And a command doesn't have a "function result".

To access the desired string the next command to be executed has to be a parameterized command in the sense that is it has to be a function that expects a string as argument, e.g.

```
FUN foo : string -> com
```

The command `readLine(stdIn)` and the function `foo` can then be combined with a variant of the "`&`"-function:

```
readLine(stdIn) & foo
```

This variant of the "`&`"-function passes the result of the first command (the string read from standard input) as argument to the function `foo`, yielding a new command. The argument string itself can be accessed in `foo` like any other parameter of a function definition.

Very often the second command will be written as a lambda-abstraction. For example, a command `echo` which echoes the input to the output can be written as

```
FUN echo : () -> com[void]
DEF echo () == readLine(stdIn)      & (\\ x.
               writeLine(stdOut, x) &
               echo() ))
```

or—if you don't like never terminating programs—as

```
DEF echo() ==
      readLine(stdIn)               & (\\ x.
        IF x =  empty THEN writeLine(stdOut, "End of Program")
        IF x |= empty THEN writeLine(stdOut, x)  &
                           echo() )
        FI
```

The unusual functionality of `echo` results from the restriction that constants cannot be cyclic, i.e. the definition of a constant cannot be recursive. Therefore `echo` must be defined as a function with an empty argument.

5.3 Error-Handling

While "`&`" does some error-handling automatically, it is also possible to do error-handling by yourself with the corresponding function "`;`". When using the function "`;`" instead of "`&`" the second command does not receive the required value directly, but an answer that contains the value if the command

has been successfully executed. If the execution has failed the answer contains an error message.

The data type answer is declared as a parameterized data type in the library structure `Com[data]` as

```
TYPE ans == okay(data:data)    -- result of successful command
            fail(error:string) -- diagnostics of failing command
```

In the example `foo` the functionality has to be modified to

```
FUN foo : ans[string] -> com
```

if you want to do error-checking by yourself.

In the example `echo` error-checking could be done like

```
DEF echo ()==
      readLine(stdIn) &
        (\\ x.
           IF x okay? THEN
              IF data(x) =  empty THEN
                        writeLine(stdOut, "End of Program")
              IF data(x) |= empty THEN
                        writeLine(stdOut, data(x))
                          & echo () FI
           IF x error? THEN
              writeLine(stdOut, "Some error has occurred")
           FI
```

Note that "`x`" now has type `ans`, whereas in the first example "`x`" has type `string`. Therefore you have to use the selector `data` to access the desired string from the answer yielded by the command.

This variant of command combination is also the proposed method for supervising commands that only produce output. Commands which don't do input (i.e. which don't forward a value to subsequent commands) have functionality `com[void]`. This means they always construct an answer of type `ans` too, but the data-item will be `void`, i.e. useless. But you can check this answer to determine if an error has occurred or not.

As an example, after executing a write-command you can check if this command was successful as follows:

```
writeLine(stdOut, "This will be the first line"!) ;   (\\ x.
      IF x okay? THEN <<<everything all right>>>
      IF x error? THEN <<<some error occurred>>>
      FI )
```

The simple version of "`;`" (as introduced at the beginning of this chapter), which combines two commands, simply ignores a failure of the first command. There is no way to check against failure, unless you use the second version of "`;`".

Chapter 6

Types in OPAL

Note: This chapter describes how to define new types in OPAL. This knowledge is not vital for trivial programs, since OPAL offers a sophisticated set of predefined types in the standard library.

A really novice user may skip this chapter altogether. Nevertheless, types and typing are fundamental for efficient and correct programming, so you should return to this chapter immediately after writing your first few programs.

OPAL is a strongly typed language with static typing which offers powerful notations for user-defined types. Each expression in a program will be associated with a unique type at compile time[1]. If this fails, a context error will result.

In the following sections we will first introduce the concept of free types, as used in OPAL, and the declaration of types (see Section 6.1). Then we will explain how to define (i.e. implement) free types (see Section 6.2).

Free types also offer an alternative way of defining functions using pattern-matching. This results in style for function definitions that is similar to term-rewriting. For details, see Section 6.3.

We will finish this chapter with some concluding remarks about parameterized types (see 6.4) and type synonyms (see 6.5).

6.1 The Concept of Free Types

Types in programming languages correspond to sets in mathematics. Thus declaring and defining (in terms of programming) means to define a set (in terms of mathematics).

[1]This associated type is described as the functionality of an expression in Section 4.3.

In OPAL there are no predefined types[2], but the standard library offers several frequently used types such as natural and real numbers, characters, strings and also more complex types such as lists, arrays, mappings, trees etc.

In this section we will introduce the concept of free types together with the *declaration* of OPAL data types. For the definition you should refer to the following section, "Definition of Types".

6.1.1 A First Example: Enumerated Types

The most simple way to define a set (in mathematics) is to enumerate all of its elements. If you need colors you may declare a type `color` by enumerating all colors:

```
TYPE color == red blue yellow green cyan orange
```

By this declaration a new set named "color" is introduced; in terms of programming, the sort "color" has been declared:

```
SORT color
```

This set consists of six elements, the colors red, blue, yellow, green, cyan and orange. In terms of programming, six constants of type "color" have been declared[3]:

```
FUN red : color
FUN blue: color
FUN yellow: color
FUN green : color
FUN cyan : color
FUN orange : color
```

And, moreover, by this declaration six discriminator functions have been declared[4]:

```
FUN red? : color -> bool
FUN blue? : color -> bool
FUN yellow? : color -> bool
FUN green? : color -> bool
FUN cyan? : color -> bool
FUN orange? : color -> bool
```

As there is no predefined equality on types (see below) these discriminator functions are the only way to distinguish between the six elements of the set "color". The function call "`blue?(x)`" yields "`true`" if and only if its argument "`x`" is evaluated as the constant "`blue`".

[2]Well, in fact there are two: booleans and denotations are actually built-ins of the compiler for obvious reasons.

[3]Don't bother about the keyword `FUN`. A constant is just a function without arguments.

[4]These discriminators have no counterpart in mathematics.

Induced Signature


The sort, the constants and the discriminator functions are called the *induced signature* of a type declaration[5]. The type declaration not only declares the new sort, but also all constants and functions of the induced signature. In the example above you have declared twelve operations and one sort in a single line! So data type declarations are a very powerful concept.

A type declaration declares by default all objects of the induced signature too. Nevertheless, you may declare them explicitly as done above. If a declaration of the induced signature is missing, the compiler will add it by itself.


In fact a type declaration is not only a declaration, although it is a specification, because it fixes the behavior of the objects of the induced signature as described above (and below).

Equality and Ordering


Note that *only(!)* the objects of the induced signature are declared by the type declaration. This means there is no equality relation, no ordering or anything else.

If, for example, you need equality of colors, you have to program it yourself[6]:

```
FUN = : color ** color -> bool
DEF = (a,b) == ((a red?) and (b red?))
               or (((a blue?) and (b blue?))
               or ....
```

This may be tedious, but because data types in Opal may contain functions and there is no computable function which can check the equality of functions, there is no way the compiler can generate an equality relation .

There is also no ordering on the elements of an enumeration type. There is no sense in saying "blue is smaller then yellow". Thus, there are no relations like "$\leq$" or ">" between elements of a type. If you need any ordering relation you have to declare (and define) it yourself.

6.1.2 A Second Example: Product Types


A very common problem is the combination of several different values in one single data object. The particulars of a person constitute a very typical example. If you want to combine the surname (`name`), the first name and a personal identification number in one object "person" you may declare:

[5]Other elements of the induced signature are constructors and selectors which will be dealt with later.

[6]Remember, you may use any identifier instead of the equal sign.

```
TYPE person == person ( name : string,
                        firstName : string,
                        id : nat)
```

Don't be worried about the two occurrences of the identifier `person`. Since OPAL supports overloading you may use different names or the same name, just as you prefer.

The first `person` (on the left-hand side of the "`==`") declares a new sort `person`, similar to the `color` example:

```
SORT person
```

The second `person` (on the right-hand side of the "`==`") may be compared with one of the concrete colors, only now it is not a constant, but a function which takes three arguments (a string, another string and a natural number) to construct a new element of the sort `person`.

```
FUN person: string ** string ** nat -> person
```

As an example, by the function call `person("Chaplin", "Charles", 1234)` the three elements "Chaplin", "Charles" and 1234 are merged into one object of type `person`.

As in the first example with colors, the declaration above also declares a discriminator function

```
FUN person? : person -> bool
```

which is rather useless in this example.

Furthermore, for each component which is combined by the constructor, there is a corresponding selector function:

```
FUN name: person -> string
FUN firstName: person -> string
FUN id: person -> nat
```

An object of type person could be decomposed by these selector functions into its elements. Let `p` be an object of type person, for example declared by

```
LET p == person("Chaplin", "Charles", 1234)
```

In this case the application of the first selector `name` on `p` will select the first element of the triple; hence, the evaluation of `name(p)` yields `"Chaplin"`. You may select the second and third element in the same way: `firstName(p)` yields `"Charles"` and `id(p)` yields `1234`.

Constructors and selectors are opposites. A constructor composes components into a single object whereas selectors decompose objects into their components or—more precisely—they select a component from a composed object.

Induced Signature


The description in the first example can now be expanded to include selectors as another part of the induced signature. The same rules apply to selectors as to the induced signature's other components (e.g. automatic declaration etc.).


Let us have a concluding note, why this kind of type declaration is called a "product type".

Imagine you have only a very small computer with a limited number of different strings and natural numbers. Say the cardinality of the set *string* is 1000 and the cardinality of the natural numbers is limited to 65536.

The set person is the three-dimensional mathematical cross-product of the set *string*, once more the set *string* and the set *nat* ($string \times string \times nat$), which means that the total cardinality of the set *person* is $1000 * 1000 * 65536 \approx 65 * 10^9$.

6.1.3 The General Concept: Sums of Products


Both examples in the previous sections are only special cases of the general concept Opal offers for declaring new types. Let us explain this concept in another example about particulars.

We are now no longer interested in the identification number, so we will omit it. But we want to know if the person is single, married, widowed or divorced. And if the person is married, we also want to know their spouse's name. On the other hand, if the person is widowed or divorced, we are not interested in the spouse's name but we do want to know the date the spouse died or the marriage was dissolved.

This could be expressed with a free type like[7]

```
TYPE person == single  ( name : string, firstName : string)
               married ( name : string, firstName : string,
                         spouse : string)
               widowed ( name : string, firstName : string,
                         dateOfDeath : date)
               divorced( name : string, firstName : string,
                         dateOfDivorce:date)
```

Before discussing the details let us summarize the signature induced by this type declaration. First of all there is the new sort `person`:

```
SORT person
```

Then we have four constructors, one for each alternative:

[7]The sort `date` must be declared somewhere, but we won't bother about it now.

```
FUN single  : string ** string -> person
FUN married : string ** string ** string -> person
FUN widowed : string ** string ** date -> person
FUN divorced: string ** string ** date -> person
```

Each constructor has a corresponding discriminator:

```
FUN single?  : person -> bool
FUN married? : person -> bool
FUN widowed? : person -> bool
FUN divorced? : person -> bool
```

Finally, there are several selectors:

```
FUN name       : person -> string
FUN firstName  : person -> string
FUN spouse     : person -> string
FUN dateOfDeath  : person -> date
FUN dateOfDivorce: person -> date
```

The type consists of four alternatives (or variants). An object of type `person` will be constructed either by the constructor `single` or `married` or `widowed` or `divorced` in the same way construction of objects was explained in the previous section (see Section 6.1.2). But in the previous section there was only one alternative. Note that the four constructors take different arguments, e.g. `married` takes three strings whereas `divorced` needs two strings and an object of type `date`.

If you have an object of type `person`, you need to know which kind of person it is, i.e. you want to know which constructor was used to compose this object. This could be tested using the discriminators as outlined in Section 6.1.1. There the discriminators were used to distinguish between the different colors, now they are used to distinguish between the different kinds of a `person`.

For example, if an object `p` is constructed by `married`

```
LET p == married(..., ..., ...)
```

then the test `married?(p)` yields `true` and `widowed?(p)` yields `false`.

You have to be able to distinguish between the four variants not only for algorithmic reasons (persons with different marital status will need different algorithmic treatment), but also for technical reasons. As the variant `married` does not contain a component about `dateOfDeath`, for example, you can't select this component from that variant. This means, assuming `p` declared as above, the function call `dateOfDeath(p)` will result in a runtime-error with program abortion. On the other hand, the component `name` is part of each alternative, so you can use this selector in all cases.

This can't be checked by the compiler, so it is the programmer's responsibility to ensure that he only uses a selector in those cases where there is also a corresponding component.

In general this will lead to a commonly employed scheme. A function with an argument of type `person` will first distinguish the variant of the argument and then do the real work:

```
DEF foo(..., p, ...) ==
        IF single?(p) THEN ...
        IF married?(p) THEN ...
        IF widowed?(p) THEN ...
        IF divorced?(p) THEN ... FI
```

This scheme could be expressed very elegantly with pattern-matching (see Section 6.3 for details).

Context Conditions


There are a few more interesting details concerning the type declaration discussed above. As you can see, it is possible to use the same selector name in different variants (e.g. `name`). The corresponding sorts need not be the same (in the case of `name`, the sort `string`) as they are just overloaded identifiers, as permitted in Opal.

Of course you may use different selector names to select "similar" components as in `dateOfDeath` and `dateOfDivorce`. But then you have to take care that the selector will only be applied to its own variant!

The ordering of variants doesn't matter at all. The ordering of the selectors is only relevant with respect to the functionality of the constructor. You may also declare the variant `divorced` as

```
...
divorced( dateOfDivorce : date, name : string, firstName : string)
```

In this case only the functionality of the constructor `divorced` changes into `FUN divorced :  date ** string ** string -> person`; everything else remains unchanged.

The names of all constructors of a type must be different. Otherwise you won't be able to distinguish between the different variants.

The name of the sort must be unique among all the sorts declared in this structure, but there is no problem using the same identifier for a sort and any function (including constructors and selectors; see the example in Section 6.1.2).

There is also no problem having the same identifier for constructors and selectors, as long as they can be distinguished by their functionality.

6.1.4 Recursive Types

In the imperative age of computing recursive types were regarded as the ultimate data types, and they were notoriously difficult to manage (dealing with pointers!).

In OPAL there is nothing magic about recursive types at all, as they fit naturally into the concept already presented.

Suppose you want to include the parents of a person in the particulars. Well, the parents are persons too, so it is very easy to declare the further enlarged type `person`:

```
TYPE person == single  ( name : string, firstName : string,
                         father: person, mother : person)
              married ( name : string, firstName : string,
                         spouse : string,
                         father: person, mother : person,
                         spousesFather: person,
                         spousesMother : person)
              widowed ( name : string, firstName : string,
                         dateOfDeath : date,
                         father: person, mother : person)
              divorced( name : string, firstName : string,
                         dateOfDivorce : date,
                         father: person, mother : person)
```

Now each object of type `person` includes the father and the mother of this person and in the case of a married person, also the parents of the spouse. Of course the induced signature changes a lot, but we won't write it down explicitly any more.

There is only one problem left. To declare a person you need the particulars person's father and mother. This means you first have to declare two objects of the sort `person` as father and mother. But to declare them you need four persons as grandparents and so on.

The nice consequence is that you finally get a whole family tree of all ancestors. The ugly consequence is that this declaration results in an endless data recursion (very similar to an endless algorithmic recursion in function definitions), because you *always* need the parents of a person to construct an object of type `person`.

In reality each person has an infinite number of ancestors, but from some point in time these are not known any more. We will model this in our type by adding a new variant `unknown`:

```
TYPE person == single  ( name : string, firstName : string,
                         father: person, mother : person)
              married ( name : string, firstName : string,
                         spouse : string,
                         father: person, mother : person,
```

```
                    spousesFather: person,
                    spousesMother : person)
          widowed ( name : string, firstName : string,
                    dateOfDeath : date,
                    father: person, mother : person)
          divorced( name : string, firstName : string,
                    dateOfDivorce:date,
                    father: person, mother : person)
          unknown
```

Now whenever we are missing information about a person we can use the constant `unknown`, which terminates the data recursion.


The notion "free type" comes from algebra. All elements of the sort `person` can be constructed using the constructors and they form a model of the "freely constructed term algebra" of the corresponding data type.


There are also examples of those data structures which can't be expressed directly by free types. A simple example is a type that includes the spouse of a person as object of type `person` again. But the spouse of the spouse is the original person. This cyclic relation can't be expressed with a free type.

6.2 Definition of Types


Definition (i.e. implementation) of types is derived straightforwardly from declaration of types (see the previous section). To implement a type, just substitute the keyword `TYPE` of a type declaration with `DATA` and you get an implementation of all objects of the corresponding induced signature:

```
DATA person == single  ( name : string, firstName : string,
                         father: person, mother : person)
               married ( name : string, firstName : string,
                         spouse : string,
                         father: person, mother : person,
                         spousesFather: person,
                         spousesMother : person)
               widowed ( name : string, firstName : string,
                         dateOfDeath : date,
                         father: person, mother : person)
               divorced( name : string, firstName : string,
                         dateOfDivorce : date,
                         father: person, mother : person)
               unknown
```

This implements a data type which fulfills all the characteristics of the corresponding free type, as described in the previous section. Therefore a type-definition is as powerful as a type-declaration.

If a corresponding free type for the sort is missing, the compiler automatically derives the free type from the definition. Therefore you can use, for example, pattern-based function definitions even without explicit declaration of a free type.

Note, however, that—in contrast to other objects—a type declaration in the signature part is *not* submitted to the implementation part. This means you have to define all objects of the induced signature (as for all objects of the signature part), but the information about being a free type is not available in the implementation part.

6.2.1 Implementation Differing from Declaration

It is also possible to have an implementation which differs from the declared free type. This is very useful in cases where you want to hide implementation details.

Imagine you want to write a structure for manipulating text. Conceptually it is a good idea to represent texts as lists of characters (similar to sequences):

```
SIGNATURE Text
IMPORT Char ONLY char
TYPE text == :: (ft:char, rt : text)
             <>
>>> a lot of additional operations<<<
```

But in practice you can't implement a text as a sequence of characters, because this implementation is slow (imagine, if you wanted to select the 5000th character in a text) and wastes enormous amounts of memory (you need additional memory at least four times as large as character stored in order to refer the next character).

Note: Although the implementation of texts as sequences is impractical, it is a very quick and reliable method for rapid prototyping of a program.

Arrays are known as fast and economical alternatives to sequences, but they are limited in size. Therefore we decided as a compromise to implement texts as a sequence (unlimited length) of arrays with fixed length (efficient):

```
DATA text == maketext(firstBlock: array[char],
                      firstFree : nat,
                      rest      : text)
             <>
/* ASSURANCE: The array component has a constant size
                   of 1024 Elements (1kB)
              The Array is filled with text up to but not including
                   the Element indexed by firstFree */
```

In this case only the objects of the induced signature of the free type corresponding to the *type definition* are defined. You as programmer are responsible

for defining all objects of the induced signature of the *type declaration*. You must ensure that your functions behave just as if they were defined by a `DATA`-definition equivalent to the `TYPE`-declaration.

In particular this means:

- The functions `<>` and `<>?` are already defined by the `DATA`-definition. But you must ensure that an empty text is *always* represented by the constant `<>`.
- The discriminator `::?` has to be defined by hand, e.g. as

```
DEF ::?(t) ==  ~(<>? (t))
```

- The selector `ft` could be defined as

```
DEF ft(t) == IF firstFree(t) > 0 THEN (firstBlock(t))[0] FI
```

- The definition of the remaining functions `::` and `rt` will be left as an exercise.

6.3 Pattern-Matching

Pattern matching is a syntactic alternative for defining functions. Instead of only giving a formal parameter name at the left-hand side of a function definition, you supply a pattern; this means this function definition will only be used if the argument matches the pattern.

Let us take the type `person` declared above as an example. Then the function `FUN knownAncestors:person->nat`, which should compute the number of ancestors including the person itself, could be defined as follows:

```
DEF knownAncestors(p AS unknown) == 0                /* definition 1 */
DEF knownAncestors(p AS single(n, fn, fa, ma)) ==    /* definition 2 */
              1 + knownAncestors(fa) + knownAncestors(ma)
```

In this case a definition only matches if the argument to the usual parameter `p` has a shape corresponding to the term behind the keyword `AS`. More specifically this pattern-based definition will be interpreted as follows:

- if the argument of a call of `knownAncestors` is the constant `unknown`, then the value is `0` (as defined by Definition 1)

- if the argument of a call of `knownAncestors` is constructed with `single`, then the arguments of the constructor can be accessed by the newly introduced parameters `n`, `fn`, `fa`, `ma`, and the value results from the right-hand side of the second definition.

- in all other cases the application of this function will result in an runtime-error, but the compiler will check whether all variants are covered and warn you beforehand. You may complete the definition by yourself.

If you don't need the parameter itself, but only the arguments of the pattern, you can omit the parameter and the keyword `AS`:

```
DEF knownAncestors(unknown) == 0                /* definition 1.1 */
DEF knownAncestors(single(n, fn, fa, ma)) ==    /* definition 2.1 */
             1 + knownAncestors(fa) + knownAncestors(ma)
```

This has a touch of term rewriting. A term on the left side (as parameter of a function definition) is substituted by the right side.

The patterns may be nested. So we can write:

```
DEF knownAncestors(single(n, fn, unknown, unknown)) == 1
                                                /* definition 3 */
```

This means that if both parents of an unmarried person are unknown, the number of ancestors is 1, just the person itself.

Note that Definition 3 does not conflict with Definition 2 as you might expect. Definition 3 is a more specialized version of Definition 2 and the program will first check whether the most specific version is appropriate and only if this fails will it use the more general one.

During compilation, pattern-based definitions are collected and transformed into one single definition with a large case distinction. Therefore in cases of ambiguity (which of two (or more) patterns is more specific) you should ensure that their definitions deliver the same results anyway.


Pattern matching can only be done on the constructors of a free type (see Section 6.1: "The Concept of Free Types"). But since a data type definition also induces a corresponding free type if the free type is missing, this is not a serious restriction. Nevertheless, pattern matching cannot be done on arbitrary functions; only constructors (and formal parameters) are allowed as elements of patterns.

It is a common error to write, e.g.

```
DEF f(0) == ...
DEF f(1) == ...
DEF f(2) == ...
DEF f(a) == ...
```

This is wrong because only the natural number 0 is a constructor, while 1 and 2 are *not*! In this case you have simply introduced local names for parameters and it is all the same whether you call them 1 and 2 or n and m or x and y.

6.3.1 Using Wildcards in Pattern-Based Definitions


In the second definition of the example in the previous section

```
DEF knownAncestors(single(n, fn, fa, ma)) ==    /* definition 2 */
            1 + knownAncestors(fa) + knownAncestors(ma)
```

the values of the parameters `n` and `fn` are never used. In this case OPAL allows the use of an underscore as a wildcard with the meaning: I know there should be a parameter, but I am not interested in its value at all.

```
DEF knownAncestors(single(_, _, fa, ma)) ==    /* definition 2.1 */
            1 + knownAncestors(fa) + knownAncestors(ma)
```

Using wildcards has the advantage that you don't need to invent new names for objects never used. This also improves the readability of the definition.

6.4 Parameterized Types


A very frequent problem in programming is to construct the same data type over different objects. A classical example for this problem are sequences. It does not matter whether you are implementing sequences of natural numbers, of characters or of persons. The algorithms and the data type will be the same in all cases. Only the sort the sequence is based on will change.

One can say the data type sequence is parameterized with the concrete basic set. In OPAL this could be expressed using parameterized structures.

For example, the parameterized data type `seq` (as included in the standard library) could be declared and implemented as:

```
SIGNATURE Seq[data]
SORT data

IMPORT Nat ONLY nat

TYPE seq == ::(ft :data, rt : seq)       /* as free type */
            <>
FUN # : seq -> nat                      /* length of a seq */
>>> a lot of additional operations<<<
```

```
IMPLEMENTATION Seq[data]

IMPORT Nat ONLY 1 +

DATA seq == ::(ft :data, rt : seq)          /* implementation of */
            <>                              /* free type         */

DEF # (s) == IF ::? (s) THEN 1 + #(rt(s))
             IF <>?(s) THEN 0 FI

>>> much more definitions<<<
```

This declares and defines a parameterized structure with a parameterized data type `seq`.

This can be used, for example, to declare a function `convert` which converts a sequence of numbers into a sequence of characters by using the structure `Seq` and instantiating the sort `seq` with the concrete sorts `nat` and `char`:

```
IMPORT Seq ONLY seq
FUN convert : seq[nat] -> seq[char]
```

We won't bother with what this function will do.

For more details about parameterization and instantiation, see Section 7.3: "Parameterized Structures and Instantiations".

6.5 No Type Synonyms

Sometimes you may want to rename a sort or use different names for the same sort. This is called type synonyms. Unfortunately, Opal does not support type synonyms.

For example, if you don't like the standard strings based on arrays, you may want to substitute them with an implementation based on the predefined structure of sequence:

```
DATA myString == seq[char]     /* illegal construction! */
```

This is not allowed in Opal (in fact it is a syntactic error). One possible circumvention is to use `seq[char]` instead of `myString` everywhere. Nevertheless, this "solution" negates your intention to explicitly distinguish between strings and sequences of characters.

The other solution is to use embedding instead of synonyms:

```
DATA myString == asMyString(asSeq : seq[char]) /* this is legal */
```

In this case you really introduce a new type and the constructor and selector only serve as type conversions or—from a different point of view—as embedding operations.

The new type `myString` doesn't inherit any functions from `seq`. Therefore you have to program all functions yourself, e.g.

```
FUN # : myString -> nat
DEF #(mS) == #(asSeq(mS))
```

where the `#` at the right-hand side of the definition is the well-known function from structure `Seq`.

Chapter 7

Programming in the Large

In contrast to programming in the small—which covers how to define single functions and data types—programming in the large describes how several functions and data types can be combined in separate units to emphasize the abstract structure of a program. These units are often called modules, structures or classes. Programming in the large also involves some other features, such as separate compilation or re usability of parts of a program.

In general, Opal follows the same principles of modular programming as other modern programming languages, e.g. Modula-2. The types and functions, etc. are collected in *structures* (similar to modules in Modula-2), which may be combined by import interfaces to form a complete program (for details, see below). Information-hiding is realized by explicit export-interfaces, analogously to Modula-2.

In the following we will describe structures and their combination by import and export interfaces (Section 7.1), how to build complete programs (Section 7.1) and we will introduce parameterized structures and their instantiation (Section 7.3).

7.1 Structures in Opal, Import and Export

In Opal the basic compilation unit is the structure. In this section we explain the relations between structures and describe how structures can be combined.

Each structure can be compiled independently from other structures. Only the export interface (i.e. the signature part) of imported structures (directly or transitively imported) is required and—provided the imported structure is not compiled already—it will also be analyzed. The compilation process is transparent for the user. For details see "A User's Guide to the Opal Compilation System"[Ma93].

As already mentioned in Chapter 2, "A First Example", a structure in OPAL consists of two parts, the signature part and the implementation part, which are physically stored in two files. A signature part only contains declarations; all definitions are delegated to the implementation part.

Objects which are declared in the implementation part are local to this structure (i.e. they can't be used in other structures) and therefore they are of no interest with respect to inter-structural relations.

7.1.1 The Export of a Structure: The Signature Part


The signature part of a structure defines the export interface of this structure. Each object declared or imported in the signature part is said to be exported by the structure. Only exported objects can be accessed from other structures by using imports (see below).


To export a data type you can declare either a free type or the objects of the induced signature in the signature part. If you don't include the type declaration in the signature part, the information about being a free type won't be included in the export interface. Therefore you can't use this information in other structures, it is not possible, for example, to use pattern-based definitions (on this data type) outside this structure. With the exception of very special applications, it is a good idea to always export the free type.


The signature part must be consistent on its own. Specifically, if a sort is used to describe the functionality of an object, this sort must be declared (maybe as part of the induced signature of a free type) or imported in the signature part too.

All imports in a signature part must be selective, i.e. you are not allowed to use complete imports (see below) in the export interface.


Furthermore, all parameterized structures which are imported in the signature part must be instantiated (for details, see Section 7.3). Uninstantiated imports are allowed only in the implementation part of a structure.

Transitive Exports


OPAL supports transitive exports. Transitive exports are objects you have imported in the *signature part* of a structure and therefore they are re-exported again by your own structure.

The consequences can best be explained in an example. Imagine you have a structure `Mystruct` which only exports the function `FUN foo : nat -> nat`. This requires the import of `nat‘Nat:SORT` (remember annotations!) in the signature part, because otherwise the signature part won't be correct:

```
SIGNATURE Mystruct
IMPORT Nat ONLY nat
FUN foo : nat -> nat
```

In this case the sort `nat` is exported by `Mystruct` too, although it's origin is the structure `Nat` and it is not declared in `Mystruct`.

When importing `Mystruct` somewhere else, you have to import `nat‘Nat:SORT` additionally, either from the structure `Nat` or—if you prefer—from structure `Mystruct`. Otherwise the sort for the argument and the result of `foo` will be missing.

7.1.2 The Import of a Structure

To use objects declared and exported by another structure you have to import them into your structure. An import can be complete or selective. In the first case you write, e.g.

```
IMPORT Nat COMPLETELY
```

This will import all exported objects of structure `Nat`. It could be regarded as enlarging your structure by the signature part of `Nat`. All objects declared in the signature part of `Nat` are now known in the importing structure too. Of course you can't redefine an imported object, because it has already been implemented in structure `Nat`.

Remember that a complete import is not allowed in the signature part of a structure.

The import could be restricted to particular objects by naming them explicitly:

```
IMPORT Nat ONLY nat + - 0 1
```

In this case only the sort `nat` and the object `+`, `-`, `0` and `1` are imported.

It is possible to have multiple imports from the same structure (and even of the same object, although this is not very useful):

```
IMPORT Nat ONLY nat + - 0 1
       Nat ONLY * /
```

It might be helpful in importing additional objects for auxiliary functions, for example, which should not appear in the main imports.

The import hierarchy must be acyclic, i.e. if a structure `Struct_a` imports a structure `Struct_b`, then `Struct_b` can't import `Struct_a`, neither directly nor transitively from other structures.

Overloaded Names in Imports

The handling of overloaded names in imports is different from the general scheme for overloading in OPAL. Whereas normally the compiler complains if it can't identify an object unambiguously, in the case of imports *all* possible objects are imported.

The import of

```
IMPORT Int ONLY int -
```

for example, imports the sort `int` and both functions `-`, the unary as well as the dyadic:

```
FUN - : int->int
FUN - : int ** int -> int
```

This generally simplifies the imports, but it can sometimes also lead to an unexpected error. Imagine a structure which exports two functions `is`:

```
SIGNATURE Mystruct
IMPORT ...
FUN is : nat -> bool
FUN is : char -> bool
```

Then you do an import

```
IMPORT Mystruct ONLY is
       Nat      ONLY nat
```

because you want to use the first function.

The compiler will complain with an error message such as "`ERROR: application of is needs import of char`", because *both* functions are imported and the second one requires the sort `char`.

If you only want to import the first function you must be more specific and annotate the imported objects:

```
IMPORT Mystruct   ONLY is:nat->bool
       Nat        ONLY nat
```

Generally it is a good idea to import only those objects which are really required by your algorithm. You should prefer selective imports to complete imports for several reasons:

- Compilation time decreases because the compiler has less objects to manage.
- The documentary expressiveness of the program text increases.

- The quality of the program increases, as the programmer is forced to think more carefully about the imports.

Nevertheless, sometimes during program development it is useful to do a complete import if you don't want to think too much about the import.

There is a tool called "browser" available which transforms a complete import into an appropriate selective import with respect to the objects really needed, see [Dz93] for details.

7.1.3 Systems of Structures

An OPAL structures can be combined in subsystems and systems, thus comprising private libraries. The handling of these "super-large" programming structures is the task of the OPAL Compilation System, not of the language OPAL itself.

For more information on how to create subsystems with several structures and use different libraries, refer to "A User's Guide to the OPAL Compilation System" [Ma93].

7.1.4 Importing Foreign Languages

Even OPAL supports the integration of modules written in foreign languages into OPAL programs. In the case of C this is called hand-coding (because C is also the target language of the compiler).

The import of foreign modules is recommended only for very special purposes. In the current release, e.g. some few structures of the library have been hand-coded due to efficiency reasons (e.g. numbers and texts) or because they need routines which depend on the operating system(e.g. access to the file system) and therefore cannot be expressed in OPAL itself.

If you are thinking about doing hand-coding by yourself, *don't do it.* If you are prepared to go to any length, you should refer to "Hand-coder's Guide to OCS Version 2" [GrSü94].

7.2 OPAL Programs

Now A complete OPAL program consists of several OPAL structures which may reside in various subsystems and in the actual working directory. There is no syntactic item which identifies the "main structure", the "entry point" or the "start function" of a program. Instead, you have to tell the OPAL compilation system (OCS) which structure should be considered the root of the program (i.e. the "main structure") and which function the entry point.

The OCS call (cf. Chapter 2: "A first Example: Hello World")

```
> ocs -top HelloWorld hello
```

will compile the structure `HelloWorld` and—as far as necessary—all imported structures of `HelloWorld` (directly imported as well as transitively imported) and then link the resulting object code.

The compilation process is optimized such that structures are only recompiled when their former compilations are obsolete because of changes in the program text. This optimized compilation is handled automatically by OCS and the user doesn't need to worry about the order of the several compilation steps. Submitting one compile command with the root structure of the program will compile all necessary structures.

The call above tells OCS to use the function `hello` as the entry point of the program and this name will also be the name of the generated executable binary file. When calling the generated binary program `hello`, the function `hello` of the root structure `HelloWorld` will be evaluated.


This entry point must be a constant of type `com[void]` and must be exported by the signature part of the root structure.

```
SIGNATURE HelloWorld
...
FUN hello : com[void]
```

There can be several functions with this functionality in the root structure, so the entry point of the program can easily be altered; but for each generated binary you must explicitly state which one should be used as entry point.

7.3 Parameterized Structures


As already mentioned in Section 6.4, a common difficulty is writing a data structure or algorithm which is more general than the usual typing restrictions allow.

Sets are a typical example. The algorithms for enlarging a set, checking if a data item is a member of the set or computing the cardinality of a set are the same whether it is a set of numbers, a set of strings or even a set of a sets of persons. In traditional programming languages (e.g. Modula-2) you would have to write the data structure set several times as set of numbers, set of characters and so on.

Opal offers the concept of parameterized structures to avoid this nasty and boring repetition. You could write the structure set once using a parameterized structure and the concrete instantiation (i.e. set of numbers, set of characters etc.) will be fixed when using the structure.

In the following we describe how to write (Section 7.3.1) and how to use (Section 7.3.2) parameterized structures.

7.3.1 How to write Parameterized Structures


Writing parameterized structures is easy. You just have to add the names of the parameters to the structure name as annotations in the signature part:

```
SIGNATURE Set[data, <]
```

You also have to declare what these parameters should be:

```
SORT data
FUN < : data ** data -> bool
```

In this example the first parameter `data` is a sort and the second "`<`" a dyadic operation which takes two elements of this sort and yields a boolean value. Of course you cannot define the objects of the parameter list (`data`, `<`) in the implementation part. But you can use them as if they were declared (and in fact they are) in the structure `Set` just like any other object.

As an example you could write a function

```
FUN foo : data ** data -> data
DEF foo(a,b) == IF a < b THEN a
                IF b < a THEN b FI
```

or declare a new type

```
TYPE okOrError == ok(value:data)
                  error(msg:string)
```

Because you can use the parameters anywhere in the structure, not only the structure as a whole but also each single function and sort is parameterized with `data` and `<`.

Remember however that `<` is just a name—nothing more—and can be substituted by any other name. In the case of the structure `Set` the intention is, that the function should be a total strict order, though this can only be expressed in comments because this is a semantic requirement that can't be checked by the compiler.

7.3.2 How to use Parameterized Structures


To use a parameterized structure you have to substitute the abstract parameters by concrete values. This is called instantiation. To use sets of natural numbers you could import

```
IMPORT Set[nat'Nat, <'Nat] ONLY set in #
```

In this import the parameter `data` is instantiated with `nat` and the parameter `<` with the function `<` from the structure `Nat`. Note that the fact that parameter and instance have same identifier (`<`) is pure chance.

The signatures of the imported objects are

```
SORT set
FUN in : set ** nat -> bool
FUN #  : set -> nat
```

Remember, in the structure `Set` the function `in` has been declared as `FUN in : set ** data ->bool`. This `data` has been substituted by `nat` due to the instantiation.

Using this import you can write a function

```
FUN foo : set -> bool
DEF foo(s) == (#(s)) in s
```

for example, which checks if the number of elements of a set is a member of the set itself.

You can also use uninstantiated imports of parameterized structures. Conceptually you import all possible instances of the structure; of course these will be quite numerous.

In the import

```
IMPORT Set ONLY set in #
```

the parameters `data` and `<` are still undefined.

Uninstantiated imports are allowed only in the implementation part, in signature parts you always have to use instantiated imports.

Note that in case of an uninstantiated or more then one instantiated import the application of `set` is still ambiguous and therefore has to be annotated to resolve this ambiguity.

If you want to declare a function `transform`, which converts a set of numbers into a set of characters, you have to instantiate the sort `set` by annotations:

```
FUN transform : set[nat, <] -> set[char, <]
```

Uninstantiated imports are useful if you need several different instantiations. In the case above you may also do two instantiated imports:

```
IMPORT Set[nat, <] ONLY set in #
IMPORT Set[char, <] ONLY set in #
```

But this doesn't help because now you have two sorts named `set` and they too have to be distinguished in the declaration of `transform` by annotations.

Appendix A

The Standard Library

Conceptually OPAL offers no built-in data types. There are no numbers, no texts, no complex data types like arrays or anything else as is customary in other programming languages.

In OPAL these issues are deferred to the standard library, which offers a large number of simple as well as powerful data structures and algorithms. At the moment there are more then one hundred structures and the number continues to increase as new features are added to the library.

In this appendix we will give a short survey of the features offered by the standard library. It won't be a complete guide, just a glance. For more detailed information you should refer to the guide "Bibliotheca Opalica" [Di94], the library itself (in subdirectory `.../lib`) and to the online documentation system "olm".

The library has a two-dimensional structure with several naming conventions. In general there are several auxiliary structures for each data type, which offer additional operations for the data type in question. These conventions are straigthforward. They are explained in [Di94] and should help to prevent the user getting lost in the huge library.

The library is divided into five subsystems: Internal, Basic Types, Functions, Aggregate Types and System. In the following sections we provide a short overview of each of these subsystems.

A.1 Internal

There are only two structures in this subsystem which are of interest to the user: `BOOL` and `DENOTATION`.

Because boolean values and denotations are essential for the compiler, they are (in contrast to all theory) in fact not realized with library structures but are built-ins of the compiler. The two structures `BOOL` and `DENOTATION` only ensure correct management during compilation.

These two structures are always imported automatically. Therefore these two structures cannot be substituted by user-written structures as it is possible for all other structures.

Note: There is a second data type (called `string`) for representation of text beside denotations; see A.4 for differences.

A.2 Basic Types


In this subsystem the customary data types are declared. This encompasses natural and integral numbers, real numbers, characters and also some additional operations for bools and denotations.

A.3 Functions


The subsystem `Functions` contains several structures which support the combination of functions in various ways, e.g. for the well-known function-composition, for iterating functions, for combining predicates or for defining ordering relations on arbitrary data types.

A.4 Aggregate Types


In the subsystem `Aggregate Types` a bunch of more or less complex data types is defined. These range from the very familiar, like `strings`, to the extremely complex, e.g. arbitrary graphs.

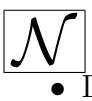
In detail, there are

- products, which realize cartesian products with up to four dimensions;
- unions, which realize disjoint unions with up to four data types;
- sequences of arbitrary elements as the most frequently used data structure in functional programming;
- strings as an alternative method for representing text; Strings could be thought of as sequences of characters, whereas denotations are more like arrays of characters. You should use denotations if the text does not change very often (e.g. for fixed messages), whereas strings are better if you are continually modifying the text.

  There are also structures to scan and to format strings.

- sets of arbitrary elements; Sets could also be defined with predicates, which describe the members of the set or as bitsets which are fast but limited in size.
- bags as sets with multiple elements;
- maps as implementations of arbitrary mappings;
- arrays as a special (and efficient) form of mappings with a domain restricted to natural numbers;
- graph-like data structures; This subsystem is still under construction. At the moment there are only AVL-trees available.

A.5 System


This subsystem consists of four part:

- Debugging offers some functions we hope you will never need;
- Commands realize the basis for input and output;
- Streams offer a simple communication protocol, which is independent form the operating system used;
- Unix supports access to UNIX-specific features like file-system, environment and processes;

Appendix B

More Examples of OPAL-Programs

This appendix will summarize some examples of OPAL programs.

B.1 Rabbits

The program Rabbits has already been presented in Chapter 1. We include it here only for the sake of completeness.

It computes the Fibbonacci-numbers and it is an example of simple interactive I/O. Also the conversions between denotations, strings and natural numbers ("‘" and "!") are worth a look.

Rabbits.sign

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

Rabbits.impl

```
IMPLEMENTATION Rabbits

IMPORT  Denotation      ONLY  ++
        Nat             ONLY nat ! 0 1 2 - + > =
        NatConv         ONLY ‘
        String          ONLY string
        StringConv      ONLY ‘
        Com             ONLY ans:SORT
        ComCompose      COMPLETELY
        Stream          ONLY input stdIn readLine
```

```
                              output stdOut writeLine
                              write:output**denotation->com[void]

-- FUN main : com[void] -- already declared in signature part
DEF main ==
    write(stdOut,
 "For which generation do you want to know the number of rabbits? ") &
    (readLine(stdIn)    & (\\ in.
     processInput(in‘)
    ))

FUN processInput: denotation -> com[void]
DEF processInput(ans) ==
          LET generation == !(ans)
              bunnys     == rabbits(generation)
              result     == "In the "
                            ++ (generation‘)
                            ++ ". generation there are "
                            ++ (bunnys‘)
                            ++ " couples of rabbits."
          IN writeLine(stdOut, result)
-- ------------------------------------------------------------

FUN rabbits : nat -> nat
DEF rabbits(generation) ==
        IF generation = 0 THEN 1
        IF generation = 1 THEN 1
        IF generation > 1 THEN rabbits(generation - 1)
                               + rabbits(generation - 2)
        FI
```

B.2 An Interpreter for Expressions

This interpreter for simple mathematical expressions was developed as an exercise for students in the first semester. After two months of programming experience they had to implement the evaluation and formatting functions, while the parser and the I/O environment were implemented by the teaching staff.

The goal was to format a simple arithmetic expression in pre- and infix-order and to compute the value of the expression.

The whole program is divided into three structures:

- `Expr`: This structure does the formatting and evaluation of expressions, i.e. this was the student's part.
- `Parser`: The parser transforms the textual input into an object of type

expression, which can be processed by the functions of the structure `Expr`. Because the implementation part is rather lengthy and not so interesting, we omit it in this paper.

- `ExprIO`: This structure controls the input and output and calls the routines of the parser, the formatters and the evaluator in an the required sequence.

B.2.1 ExprIO.sign

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

B.2.2 ExprIO.impl

```
IMPLEMENTATION ExprIO

IMPORT  String          ONLY string  ! ++ slice   = |=  % # empty
        Char            ONLY char = |= space? newline

        Com             ONLY ans:SORT okay? data fail?
        ComCompose      COMPLETELY
        Stream          ONLY input stdIn readLine
                             output stdOut write writeLine
        Nat             ONLY nat + - 0 1 2 3 4 5 6 7 8 9 10 * = |=  >
        Expr            COMPLETELY
        Parser          COMPLETELY

DEF interpreter ==
        writeLine(stdOut,
   "Bitte einen mathematischen Ausdruck eingeben, z.B: 4+(3^2)!-2") &
       (readLine(stdIn) & (\\ in.
        processInput(in) )


FUN processInput: string -> com[void]
DEF processInput(ans) ==
   IF ans = empty THEN writeLine(stdOut, "Ende des Programms"!)
   IF ans |= empty THEN
      LET res == parse(ans)
      IN
      IF res error? THEN
        writeLine(stdOut, msg(res)) &
        (writeLine(stdOut,
```

```
                "Bitte noch einmal versuchen (leere Eingabe -> Ende):")&
        (readLine(stdIn) &
         processInput))
      IF res ok? THEN
        doWork(exprOf(res)) &
        (writeLine(stdOut,
                 "Naechster Ausdruck (leere Eingabe -> Ende):")&
        (readLine(stdIn)&   (\\ in.
         processInput))
      FI
   FI



FUN doWork : expr -> com[void]
DEF doWork(e) ==
  writeLine(stdOut, "Der Ausdruck in Prefix-Notation ist:")&
  (write(stdOut, ">>>")&
  (write(stdOut, prefix(e))&
  (writeLine(stdOut, "<<<")&
  (writeLine(stdOut, "Der Ausdruck in Infix-Notation ist:")&
  (write(stdOut, ">>>")&
  (write(stdOut, infix(e))&
  (writeLine(stdOut, "<<<"!)&
  (write(stdOut, "Der Wert des Ausdrucks ist: "!)&
  (writeLine(stdOut, eval(e))))))))))
```

B.2.3 Parser.sign

```
SIGNATURE Parser

IMPORT  Expr    ONLY expr
        String  ONLY string

TYPE parseRes == ok(exprOf : expr)
                 error(msg : denotation)

FUN parse : string -> parseRes
```

B.2.4 Expr.sign

```
-- ---------------------------------------------------------------
--      Textual Representation and Evaluation of Expressions
--
-- ---------------------------------------------------------------
```

```
SIGNATURE Expr

IMPORT Nat ONLY nat

TYPE  expr ==   number (val: nat)
                dyadic (left: expr, dyop: dyadicOp, right:expr)
                monadic(monop : monadicOp, exprof :expr)

TYPE dyadicOp == addOp multOp minusOp divOp powerOp

TYPE monadicOp == facOp

-- ---------------------------------------------------------------

FUN prefix : expr -> denotation
FUN infix : expr -> denotation
FUN eval : expr -> denotation
```

B.2.5 Expr.impl

```
-- ---------------------------------------------------------------

IMPLEMENTATION Expr

IMPORT Nat ONLY = > 0 1 + - * /
IMPORT Denotation  ONLY ++
IMPORT NatConv ONLY ‘

DATA  expr ==   number (val: nat)
                dyadic (left: expr, dyop: dyadicOp, right:expr)
                monadic(monop : monadicOp, exprof :expr)

DATA dyadicOp == addOp multOp minusOp divOp powerOp

DATA monadicOp == facOp

-- ---------------------------------------------------------------

DEF prefix(number(e)) == e‘
DEF prefix(monadic(op,e)) ==  monop2textPrefix(op) ++
                                "(" ++ prefix(e) ++ ")"
DEF prefix(dyadic(l,op,r)) == dyop2textPrefix(op) ++
                              "(" ++ prefix(l) ++ ", " ++
                              prefix(r) ++ ")"
```

```
DEF infix(number(v)) == v‘
DEF infix(monadic(op,e)) == "(" ++ infix(e) ++ ")"
                              ++ monop2textInfix(op)
DEF infix(dyadic(le,ope,re)) ==
               LET l ==  "(" ++ infix(le) ++ ")"
                   r ==  "(" ++ infix(re) ++ ")"
                   op == dyop2textInfix(ope)
               IN l ++ op ++ r

-- Auxiliary functions for textual representation

FUN monop2textPrefix : monadicOp -> denotation
DEF monop2textPrefix(op) ==
        IF op facOp? THEN "fac"
        FI

FUN dyop2textPrefix : dyadicOp -> denotation
DEF dyop2textPrefix(op) ==
        IF op addOp? THEN "add"
        IF op minusOp? THEN "minus"
        IF op multOp? THEN "mult"
        IF op divOp? THEN "div"
        IF op powerOp? THEN "pow"
        FI


FUN monop2textInfix : monadicOp -> denotation
DEF monop2textInfix(op) ==
        IF op facOp? THEN "!"
        FI


FUN dyop2textInfix : dyadicOp -> denotation
DEF dyop2textInfix(op) ==
        IF op addOp? THEN "+"
        IF op minusOp? THEN "-"
        IF op multOp? THEN "*"
        IF op divOp? THEN "/"
        IF op powerOp? THEN "^"
        FI

-- ---------------------------------------------------------------
-- evaluation of expressions

DATA result == ok(val:nat)
```

```
                error(msg:denotation)

FUN ok : nat -> result
    error : denotation -> result

FUN val : result -> nat
    msg : result -> denotation

FUN ok? error? : result -> bool



DEF eval(e) ==
        LET res == eval1(e)
        IN
        IF res ok? THEN val(res)‘
        IF res error? THEN  msg(res)
        FI

-- ---------------------------------------------------------------------
-- Bei der Fehlermeldung werden alle gefundenen Fehler gemeldet

FUN eval1 : expr -> result
DEF eval1(e) ==
        IF e number? THEN ok(val(e))
        IF e monadic? THEN evalMonadic(monop(e), eval1(exprof(e)))
        IF e dyadic? THEN evalDyadic(dyop(e),
                                     eval1(left(e)),
                                     eval1(right(e)))
        FI

FUN evalMonadic : monadicOp ** result -> result
DEF evalMonadic (op, arg AS error(msg)) == arg
DEF evalMonadic (op, arg AS ok(val))    == ok( doMonop(op)(val))

FUN doMonop : monadicOp -> nat -> nat
DEF doMonop(op) ==
        IF op facOp? THEN fac
        FI


FUN evalDyadic : dyadicOp ** result ** result -> result
DEF evalDyadic (op, arg1, arg2) ==
        IF (arg1 error?) and (arg2 error?)
                   THEN error(msg(arg1) ++ msg(arg2))
```

```
        IF (arg1 error?) and (arg2 ok?) THEN arg1
        IF (arg1 ok?) and (arg2 error?) THEN arg2
        IF (arg1 ok?) and (arg2 ok?) THEN
          IF (op divOp? ) and (val(arg2)=0)
                   THEN error("Division durch Null ")
          IF (op minusOp? ) and (val(arg2)> val(arg1))
                          THEN error("Minuend kleiner Subtrahend ")
          IF (op powerOp?) and ((val(arg1)=0) and (val(arg2)=0))
                          THEN error("Basis und Exponent gleich Null")
           ELSE ok(doDyop(op)(val(arg1), val(arg2)))
          FI
        FI

FUN doDyop : dyadicOp -> nat ** nat -> nat
DEF doDyop(op) ==
        IF op addOp? THEN +
        IF op minusOp? THEN -
        IF op multOp? THEN *
        IF op divOp? THEN /
        IF op powerOp? THEN pow
        FI

-- ---------------------------------------------------------------
FUN pow : nat ** nat -> nat
DEF pow(x, y) == IF y = 0 THEN 1
                 IF y = 1 THEN x
                 IF y > 1 THEN x * pow(x, y-1)
                 FI
-- ---------------------------------------------------------------------
FUN fac : nat -> nat
DEF fac(x)==    IF (x =0) or (x=1) THEN 1
                IF x > 1 THEN x * fac(x-1)
                FI
```

B.3 An Arbitrary Directed Graph

Sorry, the directed Graph is not available yet.

Appendix C

Common Errors and What To Do

This appendix will describe some common programming errors and how to avoid them. The collection is incomplete as we are still looking for new errors. If you have found a typical programming error and you think it should be included here, we would encourage you to send this error (together with the solution) to the author (email: jue@cs.tu-berlin.de).

1. **Error Message:** `Expected was FI ...  instead of OTHERWISE`
   Possible reason: Behind an `OTHERWISE` there must be another guard; no immediate object declaration (`LET ...`) is allowed.

2. **Error Message:** `Expected was IN ...  instead of IF`
   Possible reason: `IF`-expressions within other expressions have to be enclosed in parantheses.

3. **Error Message:** `Missing Operand`
   Possible reason: Missing delimiter (e.g. space) between graphical symbols; Example: `a::b::c::<>` must be written as `a::b::c::  <>`

4. **Error Message:** `Undefined identification`
   Possible reason: Typically wrong typing: missing import of the operation, import with the wrong functionality (in cases of overloaded functions), wrong functionality in declaration.

5. **Error Message:** `Undefined identification`
   Possible reason: Function application in prefix notation without paranthesss, compiler will assume postfix notation: `% c` must be written as `%(c)`

Bibliography

[Pe91] P. Pepper: *The Programming Language* Opal; Technical Report No. 10-91, FB Informatik, TU Berlin; third corrected edition Dec. 92

[SchGr92] W. Schulte and W. Grieskamp: *Generating Efficient Portable Code for a Strict Applicative Language* in: Proceedings of Phoenix Seminar and Workshop on Declarative Programming; Springer 1992

[Di94] K. Didrich: *Bibliotheca Opalica*; TU Berlin, 1994

[Ma93] Ch. Maeder: *A User's Guide to the* Opal *Compilation System*; TU Berlin, 1993

[Le94] A. Lennartz: *Entwurf und Implementierung eines Pseudo-Interpreters für die Programmiersprache* Opal; Thesis, TU Berlin, 1994

[Dz93] R. Dziallas: *Entwicklung von Anzeigefunktionen für analysierte* Opal*Quellen*; Thesis, TU Berlin, 1993

[GrSü94] W. Grieskamp and M. Südholt: *Handcoder's Guide to OCS Version 2*; TU Berlin, 1994