MORPHO MANUAL
MORPHO VERSION 2

Snorri Agnarsson
Department of Computer Science, University of Iceland, 101 Reykjavík, Iceland.
snorri@hi.is

October 8, 2019

The Morpho system can be downloaded from
`http://morpho.cs.hi.is`.

Morpho is a multi-paradigm programming language that supports procedural, object-oriented and functional programming. Morpho has a unique module system that is designed to make it easy to compose modules is various ways. Module operations support generic programming by viewing modules as substitutions. The Morpho linker is an integral part of the Morpho compiler.

A major design goal in Morpho is to support ultra-lightweight threads and fibers. Morpho also has closures and nested lexical environments.

The default parameter passing mechanism in Morpho is pass by value, but Morpho also supports pass by reference, pass by name and lazy arguments.

The Morpho interpreter runtime is implemented using a very simple scheme of indirect threading where each operation is an object. Morpho is implemented as a scripting language on top of Java. Morpho acts as a lightweight scripting language, adding scalability and functionality to Java.

Contents

Part I.

The Morpho Language

1. Introduction

1.1. Installing Morpho

The Morpho system can be downloaded from the web[1]. Follow the instructions on the web page.

1.2. Changes from Version 1

This manual applies to version 2 of Morpho. The programming language has been extended from version 1.

- Array literals have been added.
- Support for key symbols has been added and these key symbols take part in module operations. Combined with the array literals this adds functionality that corresponds to constructors in programming languages such as ML and Haskell, but in a way that supports generic modules.
- The language now support not only call by value function arguments but also call by reference, call by name, and lazy arguments. The call by name feature allows the programmer to implement new control structures in a very simple fashion. This is used to implement database functionality that interfaces with JDBC in a convenient way.

The runtime for the programming language has been largely rewritten from version 1.

- The runtime stack is now a branching cactus stack of variables, which makes support for closures, fibers and tasks considerably faster and uses less memory. The runtime is approximately two times faster than version 1, and can run approximately four times as many concurrent fibers and tasks.
- The design philosophy behind the runtime is now based on continuations instead of being based on stateful fibers. This makes it possible to offer new functionaly for forking fibers in a simple fashion as well as making the fiber functionality simpler and easier to describe.
- The Morpho assembly language is now simpler and it is now easier to use the assembly language as a target for compilers for other programming languages[2].

[1] http://morpho.cs.hi.is

[2] The Morpho assembly language is used as a target language for compilers written in a compiler design course at the University of Iceland.

1.3. Getting Started With Morpho

The basic Morpho system consists of the following parts:

- The file `morpho.jar` contains:
  - The Morpho compiler.
  - The Morpho runtime.
  - The Morpho basis module.
- The file `Morpho.pdf` contains the Morpho documentation.

In addition there is available a folder containing code examples and extra modules.

1.3.1. The Morpho Workbench

The Morpho workbench is a simple programming environment that supports a large subset of the Morpho programming language. The workbench makes it possible to program interactively in Morpho, but is not intended for creating Morpho modules or standalone Morpho executables.

To invoke the Morpho workbench you can use the following command on Windows, Linux or Macintosh:

```
java -jar morpho.jar
```

Or:

```
java -cp morpho.jar is.hi.cs.morpho.Morpho
```

Or, assuming that the proper bat-file or shell script has been set up, you can use the command:

```
morpho
```

On Windows, you can also use:

```
javaw -jar morpho.jar
```

Or:

```
javaw -cp morpho.jar is.hi.cs.morpho.Morpho
```

The Morpho workbench is largely self-explanatory once you start it.

```
"hello.mexe" = main in
{{
main =
  fun()
  {
        writeln("Hello world!");
  };
}}
*
BASIS
;
```

Figure 1.1.: Hello World Program

1.3.2. A Simple Example

Figure 1.1 shows a simple stand-alone *hello world* program in Morpho. This is a program that is intended to be compiled and run from the command-line, not from inside the Morpho workbench.

We would type this program into a file named `hello.morpho` (or you could use any other file name if you wish). To compile the program you use the following command (assuming that your system contains the proper bat-file or shell script):

```
morpho -c hello.morpho
```

Or you can use the following command:

```
java -jar morpho.jar -c hello.morpho
```

Or:

```
java -cp morpho.jar is.hi.cs.morpho.Morpho -c hello.morpho
```

This will compile the program and create the file `hello.mexe`, which contains the executable version of the program. To run the program, we then use the command:

```
morpho hello
```

Or the following command:

```
java -jar morpho.jar hello
```

Or:

```
java -cp morpho.jar is.hi.cs.morpho.Morpho hello
```

That will cause the line

```
Hello world!
```

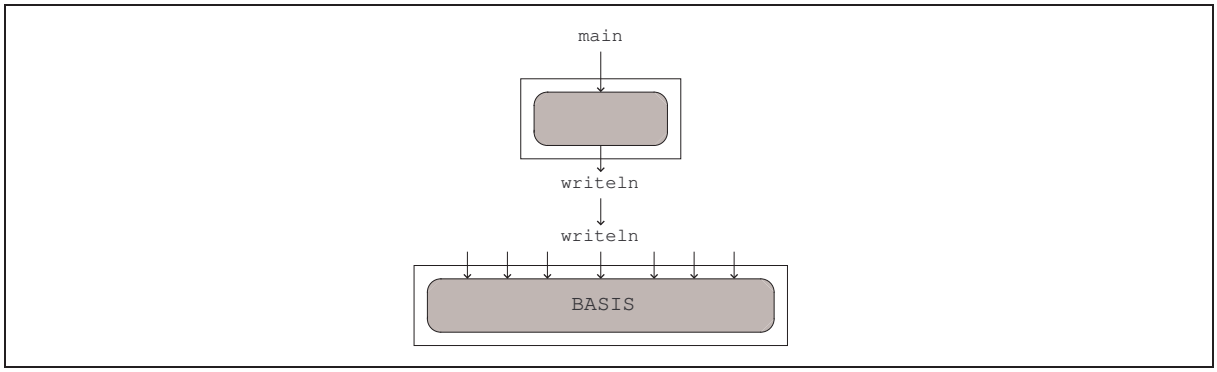


Figure 1.2.: Linking A Simple Program

to be written to standard output, i.e. the computer screen.

This program consists of two modules, linked together using the importation module operation (denoted by `*`). One of the modules contains the function `main` and refers to the function `writeln`. The other module is the builtin module `BASIS`, which contains a variety of built-in functions, among them being the function `writeln`.

Figure 1.2 shows how the above program is linked. Further on in this manual we will see more module operations.

1.3.3. Another Example

The Morpho program in figure 1.3 on the following page is a little bit more complex.

As before, we might type this program into a file named `fibo.morpho`. To compile the program you could then use the command:

```
morpho -c fibo.morpho
```

This will compile the program and create the file `fibo.mexe`, and to run the program, we then use the command:

```
morpho fibo
```

That will cause the line

```
fibo(10)=55
```

to be written to standard output, i.e. the computer screen.

This program consists of three modules, linked together using a couple of different module operations, the importation module operation (denoted by `*`), and the iteration operation (a unary module operation denoted by `!`). One of the modules contains the function `main` and refers to the functions `writeln`, `++` and `fibo`. The other module is the builtin module `BASIS`, which contains a variety of built-in functions, among them being the functions `++`, `writeln`, `<=`, `-`, and `+`.

Figure 1.4 on page 13 shows how the above program is linked. Remember that further on in this manual we will see more module operations.

```
"fibo.mexe" = main in
{{
main =
  fun()
  {
        writeln("fibo(10)="++fibo(10));
  };
}}
*
!
{{
fibo =
  fun(n)
  {
    if( n<=2 )
    {
      return 1;
    }
    else
    {
      return fibo(n-1)+fibo(n-2);
    };
  };
}}
*
BASIS
;
```

Figure 1.3.: Fibonacci Program

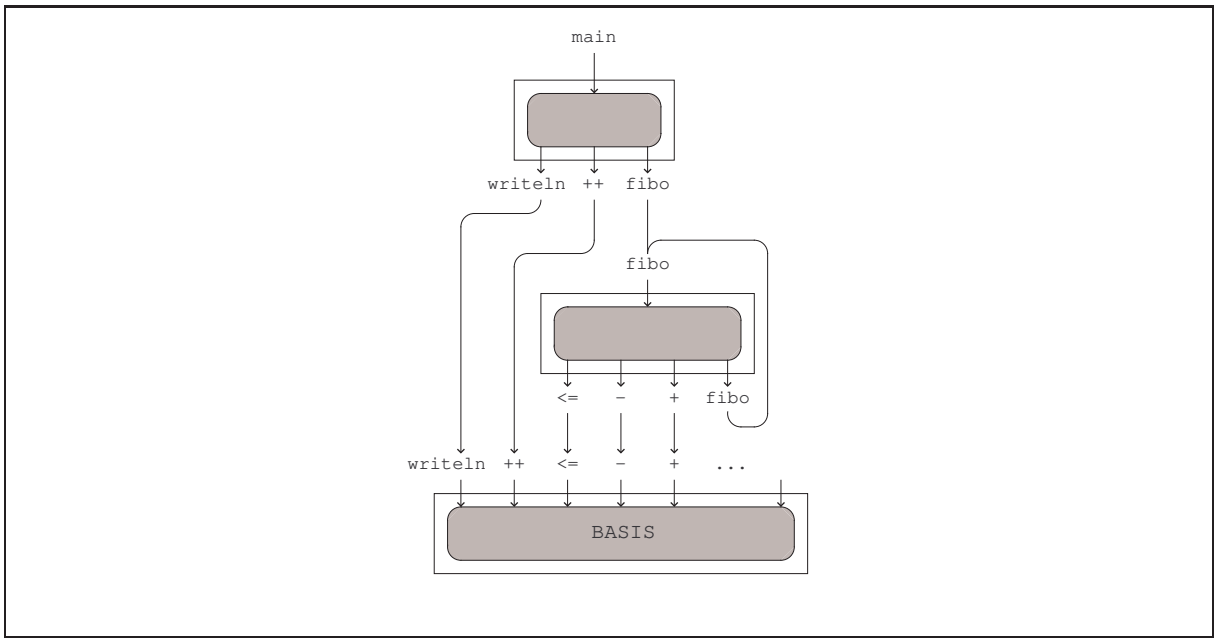


Figure 1.4.: Linking The Fibonacci Program

2. Expressions

Morpho has a wide variety of expression but no statements. Constructs such as if-then-else and while-do that are defined as statements in many other programming languages are expressions in Morpho.

2.1. Morpho Values

All values in Morpho are references to Java objects.[1]

2.2. Literals

Literal expressions in Morpho evaluate to integers, doubles, characters, boolean values, strings, and the null reference. Evaluating a literal has no side effect.

2.2.1. Integer Literals

The syntax of integer literals is defined by syntax diagram 34 on page 57. Integer literals can be any number of digits. The value of an integer literal is a corresponding Java Integer object, a Java Long object or a Java BigInteger object. There is no limit on the size of an integer literal.

2.2.2. Double Literals

The syntax of double literals is defined by syntax diagram 36 on page 58. The value of a double literal is a corresponding Java Double object.

2.2.3. String Literals

The syntax of string literals is defined by syntax diagram 38 on page 59. The value of a string literal is a corresponding Java String object.

[1] Future versions and variants of Morpho may run on top of other platforms than Java, in which case the Morpho values may be of other types. However, the basics of the Morpho language and the Morpho literal values will remain the same.

2.2.4. Character Literals

The syntax of character literals is defined by syntax diagram 37 on page 58. The value of a character literal is a corresponding Java Character object.

2.2.5. Boolean Literals

The literals **`true`** and **`false`** are boolean literals. The value of a boolean literal is a corresponding Java Boolean object.

2.2.6. The null Literal

The literal **`null`** is the null literal. Its value is the Java null reference.

2.3. Binary and Unary Operations

Unary and binary operations in Morpho are function calls to functions with one and two arguments, respectively.

The expression '`1+2`' is evaluated by calling the function '`+`' with arguments `1` and `2`. Assuming that the function '`+`' is the usual addition function, the result will be the integer value `3`.

Similarly, the expression '`"abc"++"def"`' is evaluated by calling the function '`++`'[2] with arguments `"abc"` and `"def"`, resulting in `"abcdef"`, assuming the usual meaning of the binary operator '`++`'.

The expression '`-(1-2)`' involves two different operations, the binary operation '`-`' and the unary operation '`-`'. The unary operation is, of course, a function of one argument. Note that the expressions '`-(1-2)`', '`-1-2`', '`-1-(2)`', and '`-(1)-2`' all have different meanings because of operator precedence and also because '`-1`' is a single integer literal whereas '`-(1)`' is a unary operator applied to an integer literal.

2.3.1. Operator Precedence

Binary operators have various different precedences but all unary operators have the same precendence which is higher that that of binary operators.

The precedence of a binary operator depends on the first letter of the operator. Table 2.1 on the following page shows the various precedences. A higher precedence operator will be applied before a lower precedence operator, unless parentheses imply otherwise.

For example, the following expression pairs are equivalent:

[2]Note that the operator name '`++`' is a perfectly valid name of a binary operation. Similarly, you might think that '`1--2`' is equivalent to '`1-(-2)`', but in fact that is not the case, '`1--2`' is an expression where the binary operator '`--`' is applied to two arguments, `1` and `2`. For the expression '`1--2`' to work you would have to define the binary operator (function) '`--`'.

[1]Note that the '`&&`' binary operator has special handling and special precedence. See section 2.6.2 on page 27.

[2]Note that the '`||`' binary operator has special handling and special precedence. See section 2.6.1 on page 27.

Predecence	First letter
7	'*', '/' or '%'
6	'+' or '-'
5	'<', '>', '!' or '='
4	'&'[1]
3	'|'[2]
2	':'
1	'?', '~' or '^'

Table 2.1.: Operator Precedence

- '`1*2+3*4`' and '`(1*2)+(3*4)`'
- '`1+2<3*4`' and '`(1+2)<(3*4)`'
- '`1<2&3>4`' and '`(1<2)&(3>4)`'
- '`1&2?3|4`' and '`(1&2)?(3|4)`'

2.3.2. Operator Associativity

Binary operators with precedence 2 (those starting with the letter ':') associate to the right. All others associate to the left.

For example, the following expression pairs are equivalent:

- '`1*2/3%4`' and '`((1*2)/3)%4`'
- '`1+2-3`' and '`(1+2)-3`'
- '`1-2/3*4+5`' and '`(1-((2/3)*4))+5`'
- '`1:2:3+4+5`' and '`1:(2:((3+4)+5))`'

2.4. Variables

Morpho has a few different types of variables:

- Local variables and function parameters store information local to a particular function call.
- Global variables contain information that needs to be directly accessible in multiple parts of a system.
- Instance variable contain information local to a particular object instance.

Internally, global variables are stored in hash tables whereas all other kinds of variables are stored in locations on a branching stack that allows variable sharing between different fibers and different tasks.

Morpho supports pointers to variables (see 2.4.4 on page 26). It is possible to create a pointer to any type of variable and use that pointer to retrieve and change the value of the variable. Pointer arithmetic (as in C and C++) is not supported, nor is it possible to create pointers to array positions.

Parameters in Morpho are always passed by value. However, Morpho has syntactic sugar that allows easy emulation of parameter passing by reference and by name, as well as lazy evaluation.

2.4.1. Local Variables And Parameters

The following declarations create and define variables `x`, `y` and `z`.

```
var x=1;
var y=x+1,z;
```

The variable `x` is initialized with the integer value 1 and `y` is initialized with the result of computing `x+1`. The variable `z` has no defined initial value.

Note that the following declaration does not work as many would expect since the computation of the expression `x+1` can not be performed until `x` exists, and `x` does not exist until after the declaration has been wholly executed.

```
var x=1,y=x+1,z;
```

However, the following works:

```
rec var x=1,y=x+1,z;
```

The scope of a declarations such as the ones above extends to the end of the enclosing block, i.e. from the preceeding '`{`' to the next enclosing '`}`'. We will later see more about such blocks.

We can also use the following declarations

```
val x=1,y=2;
rec val z=x+1,w=2*z;
```

The difference between the first (**`var`**) versions and the second (**`val`**) versions is that **`var`** declarations define variables which can later receive new values in assignments such as

```
x=10;
y=x+20;
```

whereas **`val`** declarations define "variables" (really constant values) which are not allowed to receive new values. Of course it makes no sense to define a **`val`** variable with no defined value so the following declaration is not allowed:

```
val z;
```

2.4.2. Global Variables

There are three types of global variables: taskvar variables, machinevar variables and globalvar variables. They differ only in the way they interact with the parallel programming aspects of the Morpho language. Each Morpho task has a distinct copy of a taskvar variable, but all tasks in the same machine share the same copy of each machinevar variable and globalvar variable. Each Morpho machine has a distinct copy of each machinevar variable, but all machines share the same copy of each globalvar variable. Each machine can have multiple tasks and each task can have multiple fibers. On the other hand a task can only be associated with one machine and a fiber can only be associated with one task.

The following declaration declares one taskvar variable, one machinevar variable and one globalvar variable.

```
taskvar x;
machinevar y;
globalvar z;
```

A global (taskvar, machinevar or globalvar) variable must be declared in each scope it is used in. It is not possible to initialize such variables in their declaration. They are automatically initialized with the **null** value when the corresponding machine or task is created.

By Name Parameters And By Name Variables

In Morpho you can write code such as

```
var i, a=makeArray(10);
var @x = @a[i];
i = 1;
a[1] = "Hello World";
writeln(x);
```

The effect of the declaration `var @x = @a[i];` is to make the variable `x` a synonym for the expression `a[i]`. In general, code of the form

```
...
var @x = @e1;
...
x = e2;
...
e3 = x;
...
```

where `e1`, `e2`, and `e3` are some valid Morpho expressions in the given context, is equivalent to

```
...
var x = makeThunk(fun(){e1},fun(z){e1=z});
...
getThunkSetter(x)(e2);
```

```
  ...
  e3 = getThunkGetter(x)();
  ...
```

assuming that the functions `makeThunk`, `getThunkSetter`, and `getThunkGetter` are the ones defined in the BASIS module. Note, however, that there are no actual calls to these routines in the code that the Morpho compiler generates for the first code segment above, so it is not necessary to connect the BASIS module in order for the code to work.

Similarly, the following code segment

```
rec fun f(@x)
{
  ...
  x = e2;
  ...
  e3 = x;
  ...
}
...
f(@e1);
...
```

is equivalent to

```
rec fun f(x)
{
  ...
  getThunkSetter(x)(e2);
  ...
  e3 = getThunkGetter(x)();
  ...
}
...
f(makeThunk(fun(){e1},fun(z){e1=z}));
...
```

as well as

```
rec fun f(@x)
{
  ...
  x = e2;
  ...
  e3 = x;
  ...
}
...
f(makeThunk(fun(){e1},fun(z){e1=z}));
...
```

and

```
rec fun f(x)
{
  ...
  getThunkSetter(x)(e2);
  ...
  e3 = getThunkGetter(x)();
  ...
}
...
f(@e1);
...
```

Similar equivalences hold for functions exported from modules. For example we can write the following program.

```
"test.mexe" = main in
!
{{
main =
  fun()
  {
    var i=0;
    WHILE(@i<5,@{writeln(i); i=i+1});
  };

WHILE =
  fun(@c,@b)
  {
    if( !c ) {return null};
    b;
    WHILE(@c,@b);
  };
}}
*
BASIS
;
```

This program writes the numbers 0 to 4 in successive lines. This program is equivalent to the following one.

```
"test.mexe" = main in
!
{{
main =
  fun()
  {
    var i=0;
```

```
    WHILE(@i<5,@{writeln(i); i=i+1});
  };

WHILE =
  fun(c,b)
  {
    if( !thunkGetter(c)() ) {return null};
    thunkGetter(b)();
    WHILE(c,b);
  };
}}
*
BASIS
;
```

Note how the parameters `c` and `b` are passed in the recursive call to `WHILE`. In general, if a function receives an argument by name, as `@x`, then the compiler assumes that the argument is a *thunk*.

Morpho also has some additional syntactic sugar for thunks. The call

```
    WHILE(@i<5,@{writeln(i); i=i+1});
```

above can also be written as follows.

```
    WHILE(@i<5){writeln(i); i=i+1};
```

Or as follows.

```
    WHILE(@i<5)
    {
      writeln(i);
      i=i+1;
    };
```

All of these variations are exactly equivalent. Note carefully that all of these are simply function calls even though some of the variations look similar to other syntactic constructs, i.e. while loops in this instance.

Further similar variations are described in the coverage of function call syntax and the syntax of message passing in Morpho.

By Reference Parameters And By Reference Variables

In Morpho you can write code such as

```
var y;
var &x = &y;
x = "Hello World";
writeln(x);
```

The effect of the declaration **var** `&x = &y;` is to make the variable `x` a synonym for the variable `y`. In general, code of the form

```
...
var &x = &y;
...
x = e2;
...
e3 = x;
...
```

where `y` is a variable, and `e2`, and `e3` are some valid Morpho expressions in the given context, is equivalent to

```
...
var x = &y;
...
*x = e2;
...
e3 = *x;
...
```

Similarly, the following code segment

```
rec fun f(&x)
{
  ...
  x = e2;
  ...
  e3 = x;
  ...
}
...
f(&y);
...
```

is equivalent to

```
rec fun f(xp)
{
  ...
  *xp = e2;
  ...
  e3 = *xp;
  ...
}
...
f(&y);
...
```

Lazy Parameters And Lazy Variables

In Morpho you can write code such as

```
var $x = $"Hello World";
writeln(x);
```

The effect of the declaration `var $x = $"Hello World";` is to make the variable `x` a synonym for the expression `"Hello World"`, but in such a way that the expression will only be evaluated at most once. In general, code of the form

```
...
var $x = $e1;
...
e2 = x;
...
```

where `e1`, and `e2` are some valid Morpho expressions in the given context, is equivalent to

```
...
var x = makePromise(fun(){e1});
...
e2 = force(x);
...
```

Similarly, the following code segment

```
rec fun f($x)
{
  ...
  e2 = x;
  ...
}
...
f($e1);
...
```

is equivalent to

```
rec fun f(x)
{
  ...
  e2 = force(x);
  ...
}
...
f(makePromise(fun(){e1}));
...
```

Lazy variables therefore stand for *promises* that are only evaluated if and when they are forced, and are then only evaluated once. Once a promise has been evaluated, the promise remembers the resulting value and simply returns that same value again if subsequent evaluation is requested.

It is not legal to make an assignment to a lazy variable or parameter.

2.4.3. Instance Variables And Morpho Objects

Morpho objects can contain instance variables. Each object has it's own copy of each instance variable. Here's an example of a Morpho module containing a Morpho object definition, which we might call a constructor.

```
"complex.mmod" =
!
{{
;;; Use:  z = makeComplex(x,y);
;;; Pre:  x and y are real numbers.
;;; Post: z is the complex number x+iy.
makeComplex =
  obj(real,imag)
  {
    val x=real, y=imag;

    ;;; Use:  a = z.x;
    ;;; Pre:  z is a complex number.
    ;;; Post: a is the real part of z.
    msg get x
    {
      x;
    };

    ;;; Use:  b = z.y;
    ;;; Pre:  z is a complex number.
    ;;; Post: b is the imaginary part of z.
    msg get y
    {
      y;
    };

    ;;; Use:  z1 = z2.+z3;
    ;;; Pre:  z2 and z3 are complex numbers.
    ;;; Post: z1 is the sum of z2 and z3.
    msg +(z)
    {
      makeComplex(x+z.x,y+z.y);
    };
```

```
35
36      ;;; Use:  z1 = z2.-z3;
37      ;;; Pre:  z2 and z3 are complex numbers.
38      ;;; Post: z1 is the difference of z2 and z3.
39      msg -(z)
40      {
41        makeComplex(x-z.x,y-z.y);
42      };
43
44      ;;; Use:  z1 = z2.*z3;
45      ;;; Pre:  z2 and z3 are complex numbers.
46      ;;; Post: z1 is the product of z2 and z3.
47      msg *(z)
48      {
49        makeComplex(x*z.x-y*z.y,x*z.y+y*z.x);
50      };
51
52      ;;; Use:  z1 = z2./z3;
53      ;;; Pre:  z2 and z3 are complex numbers.
54      ;;; Post: z1 is the quotient of z2 and z3.
55      msg /(z)
56      {
57        val d = z.x*z.x+z.y*z.y;
58        makeComplex((x*z.x+y*z.y)/d,(y*z.x-x*z.y)/d);
59      };
60    };
61  }}
62  ;
```

The definition

```
    val x=real, y=imag;
```

defines two instance variables, `x` and `y`, which receive their values from the parameters `real` and `imag`. We could, instead, have defined the variables using the line

```
    var x=real, y=imag;
```

which would have defined the same two variables as modifiable variables instead of as constants.

The instance variables of an object are only directly accessible inside methods for that object, such as the methods for the messages `+`, `-`, `*`, and `/`. But note that we can, if we wish, define accessor messages (and their corresponding methods) for each variable, such as the ones defined for `x` and `y` in the above example.

2.4.4. Pointers To Variables

Morpho supports pointers with semantics similar to that of C and C++. If `x` is a variable of any kind (local, instance, etc.) then the expression `&x` returns a pointer to that variable. If a variable `p` contains such a pointer then the expression `*p` refers to the variable `x` and can be used as an assignment target to give `x` a new value or it can be used to fetch the value of `x`.

For example, the following lines of code will print the value 123 twice.

```
1    var x, p;
2    p = &x;
3    x = 321;    ;;; Overridden by the next line
4    *p = 123;
5    writeln("x="++x);
6    *p = 321;   ;;; Overridden by the next line
7    x = 123;
8    writeln("x="++x);
```

In contrast to the semantics of C and C++, pointer values in Morpho will remain valid indefinately. This means that variables referred to by pointers remain living while any such pointers refer to them. In particular, if a pointer is made to refer to a local variable in a function then that variable will exist after the call to the function has completed and until no pointers refer to it. Similarly, if a pointer refers to an instance variable of an object, the object will remain in existence even if there are no other references to it.

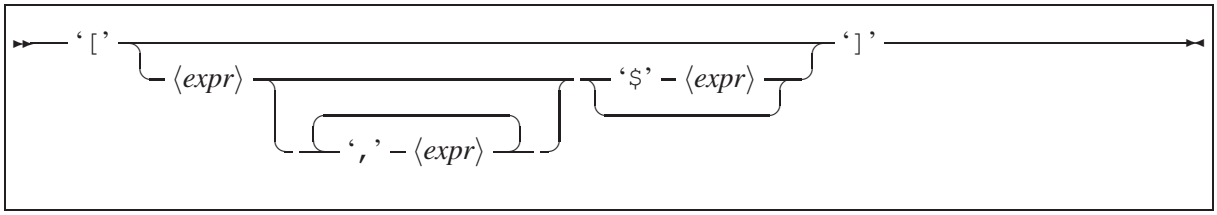
Morpho does not support pointers to positions in arrays. Nor does it support pointer arithmetic.

2.5. Lists

Morpho is, among other things, a list processing language. Morpho has built in syntax and functions to make list processing easier. Here's some example code that uses the list syntax and fundamental list functions to construct lists.

```
1    var x, y, z;
2    x = [1,2];              ;;; Construct a list
3    y = 1:2:[];             ;;; The same list again
4    z = [1$[2$[]]];           ;;; And same again
5    writeln(head(x));         ;;; Writes 1
6    writeln(head(tail(x)));   ;;; Writes 2
```

More examples of list processing can be seen in the interactive tutorial that shows up if Morpho is run in interactive mode. Syntax diagram 1 on the following page describes how to construct a list using Morpho syntax. The list syntax diagram is part of diagram 65 on page 72 for simple expressions. This list syntax is 'syntactic sugar', the same effects can be achieved by using the ':' operation in the BASIS module. An expression `[x,...]` is equivalent to the expression `x:[...]` and an expression `[x$y]` is equivalent to `x:y`.



Syntax Diagram 1: ⟨*list*⟩

2.5.1. The Empty List

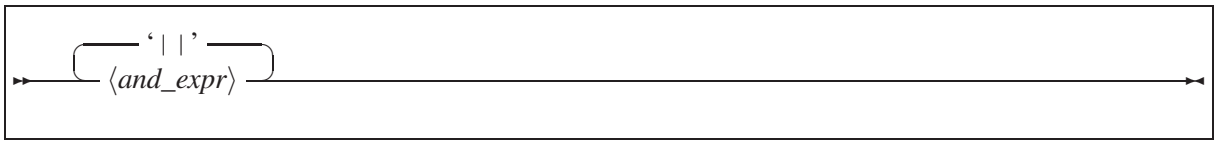
The expression `[]` is equivalent to the expression **`null`**. Both return the null reference.

2.6. Logical Expressions

Morpho has logical operators `||`, `&&` and `!`, standing for the logical operations 'or', 'and' and 'not'. Contrary to all other operations in Morpho, these are built into the language, and their semantics can not be modified. The binary logical operations have short-circuit semantics, as described below.

2.6.1. Or Expression

An or expression is a sequence of expressions separated by the operator '`||`', as shown in diagram 2 (the same diagram as 60 on page 70), The effect of an expression `x || y`, where

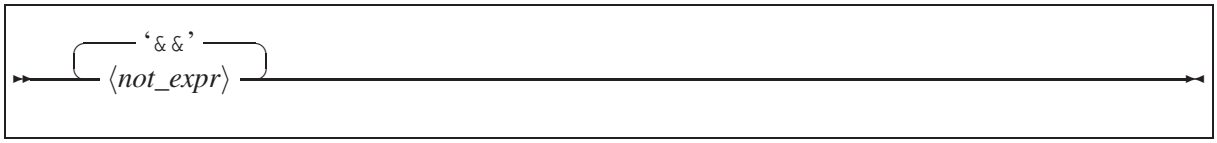


Syntax Diagram 2: ⟨*or_expr*⟩

`x` and `y` are sub-expressions, is to evaluate `x`, check whether the result is true, and if so, return that result. If the result is false then `y` is evaluated and the resulting value is the value of the whole expression. It should be noted that the only false values are the null reference and the boolean false value (the results from the expressions **`null`** and **`false`**, respectively). A consequence of this definition is that the '`||`' operator is associative, i.e. the expressions `x || (y || z)` and `(x || y) || z` are semantically equivalent.

2.6.2. And Expression

An and expression is a sequence of expressions separated by the operator '`&&`', as shown in diagram 3 on the following page (the same diagram as 61 on page 70), The effect of an expression `x && y`, where `x` and `y` are sub-expressions, is to evaluate `x`, check whether the result is false, and if so, return that result. If the result is true then `y` is evaluated and the resulting value is the value of the whole expression. As with the or expression, a consequence

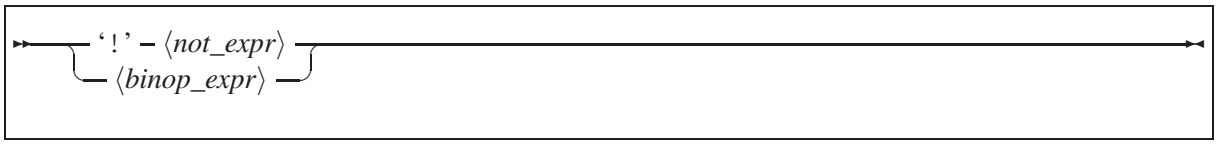


Syntax Diagram 3: ⟨*and_expr*⟩

of this definition is that the '`&&`' operator is associative, i.e. the expressions `x && (y && z)` and `(x && y) && z` are semantically equivalent.

2.6.3. Not Expression

The built-in unary operator '`!`' is the logical negation operator. Syntax diagram 4 (same as 62 on page 71) shows the pattern for not expressions. The expression `!x` returns **`true`** if

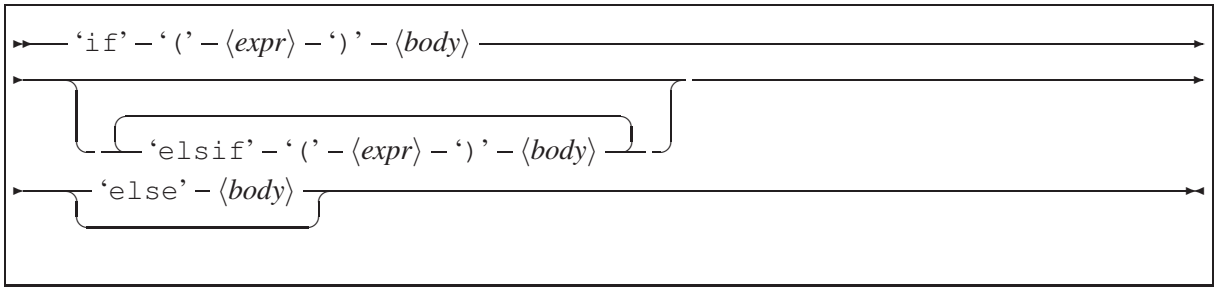


Syntax Diagram 4: ⟨*not_expr*⟩

`x` returns false and **`false`** if `x` returns true. The return value is always a boolean value but the argument to the operator can be any value. Remember that the only false values are **`false`** and **`null`**, so that `!`**`null`** as well as `!`**`false`** will return **`true`** whereas (for example) `!0` and `!"false"` will return **`false`**.

2.7. The 'if' Expression

Syntax digram 5 (same as diagram 72 on page 74) shows the pattern of the if expression in Morpho. The effect of a simple if expression



Syntax Diagram 5: ⟨*if_expr*⟩

```
1       if( cond )
2       {
3         body1;
4       }
5       else
```

```
6       {
7         body2;
8       }
```

is to evaluate the condition `cond` and then either evaluate `body1` or `body2`, depending on whether `cond` evaluated to true or false, respectively. An expression

```
if( cond )
{
  body1;
}
```

without any **`else`** part, is equivalent to

```
if( cond )
{
  body1;
}
else
{
  random;
}
```

where `random` is some undefined value.

An expression

```
if( cond1 )
{
  body1;
}
elsif( cond2 )
...
```

is equivalent to

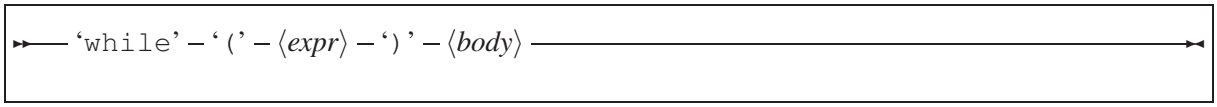
```
if( cond1 )
{
  body1;
}
else
{
  if( cond2 )
  ...;
}
```

2.8. Looping Expressions

Morpho has a couple of looping expressions; the while expression and the for expression.

2.8.1. The 'while' Expression

Syntax diagram 6 which part of syntax diagram 65 on page 72 defines pattern of the while expression. The effect of the while expression **`while`**`(c){x;}` is to evaluate the condition `c`

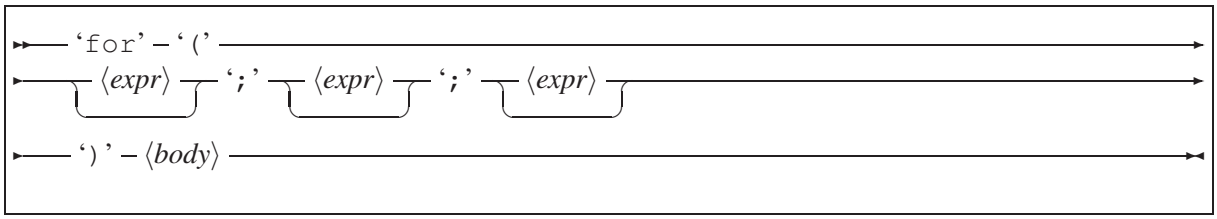


Syntax Diagram 6: ⟨*if_expr*⟩

and then if the result is true evaluate the body `{x;}`, after which the loop starts again. If the result of evaluating the condition `c` is false then the evaluation of the while expression terminates and the expression returns some undefined value (unless a break expression terminates the while expression, see section 2.8.3 on the following page).

2.8.2. The 'for' Expression

Syntax diagram 7 (same as diagram 71 on page 74) shows the syntax pattern of for expressions. The effect of a for expression of the form



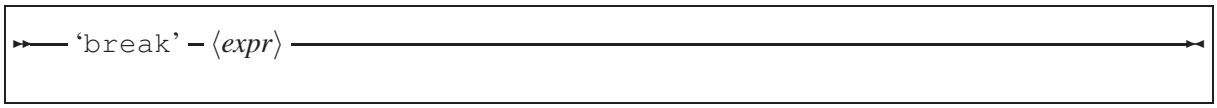
Syntax Diagram 7: ⟨*for_expr*⟩

```
for( init ; cond ; iter )
{
  body;
}
```

is as follows. First the expression `iter` is evaluated. This is done once at the beginning of the evaluation of the for expression. After that we enter a loop where the condition `cond` is evaluated each time the loop starts. If the condition evaluates to a false value (**`false`** or **`null`**) then the for expression terminates and returns some unspecified value (unless a break expression terminates the for expression, see section 2.8.3 on the next page). If the condition evaluates to a true value then the body of the expression (`body`) is evaluated and then the iteration expression (`iter`), after which we are again at the beginning of the loop. Any or all of the three expressions, `init`, `cond`, and `iter` can be omitted. If `init` is omitted then no initialization is performed. If `cond` is omitted then the condition **`true`** is used. If `iter` is omitted then nothing is done after each iteration except jumping straight to the beginning of the next loop (i.e. the evaluation of `cond`).

2.8.3. The 'break' Expression

The break expression is used to terminate a loop from within the body of the loop. Consequently, the break expression is only valid when nested in a loop. Diagram 8 (also part of diagram 59 on page 70) shows the pattern of the break expression. The effect of a break ex-

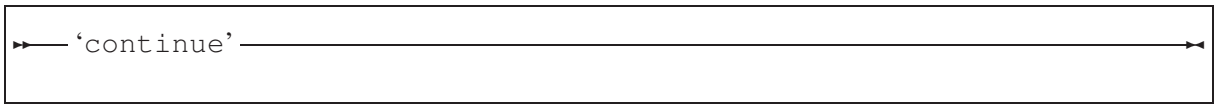


Syntax Diagram 8: ⟨*break_expr*⟩

pression **`break`** `e` is to evaluate the sub-expression `e`, terminate the current loop, and return the value from `e` as the value of the loop expression, i.e. the 'while' or 'for' expression.

2.8.4. The 'continue' Expression

The continue expression is used inside bodies of loops to go to the next iteration of the loop. Consequently, the continue expression is only valid when nested in a loop. Diagram 9 (also part of diagram 65 on page 72) shows the pattern of the continue expression. The effect of



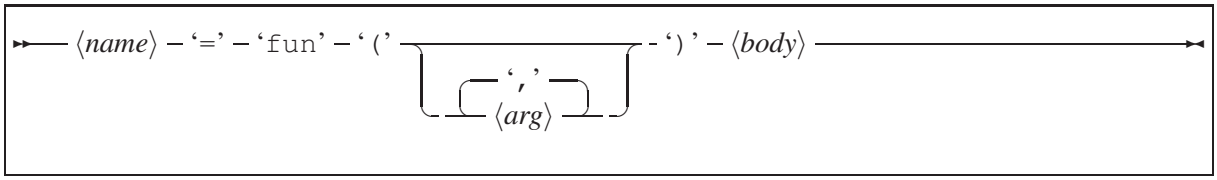
Syntax Diagram 9: ⟨*continue_expr*⟩

evaluating **`continue`** is to abandon the evaluation of the current iteration of the body of the loop and go back to the beginning, i.e. to the evaluation of the condition at the beginning of the loop.

2.9. Function Definitions

2.9.1. Top-Level Functions

Diagram 10 shows how to define a top-level function that is exported from a module (see also diagram 50 on page 66 and diagram 48 on page 65). Here are a couple of simple function



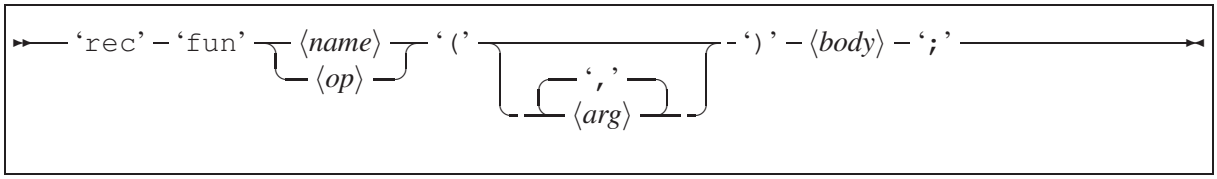
Syntax Diagram 10: ⟨*top_fun_def*⟩

definitions inside a module.

```
"fibo.mmod" =
!
{{
fibo1 =
  fun(n)
  {
    if( n<2 )
    {
      1;
    }
    else
    {
      fibo1(n-1)+fibo1(n-2);
    };
  };

fibo2 =
  fun(n)
  {
    var i = 0, f1 = 1, f2 = 1;
    while( i!=n )
    {
      val nextf = f1+f2;
      i = inc(i);
      f1 = f2;
      f2 = nextf;
    };
    f1;
  };
}}
;
```

2.9.2. Inner Functions

Diagram 11 shows how to define a single inner function that can be defined inside another function or inside an object definition. See also diagram 56 on page 69 for a fuller syntax of recursive definitions in general. An inner function definition is a declaration and can be



Syntax Diagram 11: ⟨*inner_fun_def*⟩

placed inside any body, including a function body or the bodies od compound expressions.

Here is an example of a simple function definition inside a Morpho program.

```
"fibo.mexe" = main in
{{
main =
  fun()
  {
    rec fun fibo(n)
    {
      if( n<2 )
      {
        1;
      }
      else
      {
        fibo1(n-1)+fibo1(n-2);
      };
    };
    writeln("fibo(10)="++fibo(10));
  };
}}
*
BASIS
;
```

It is also possible to define anonymous functions as stand-alone expressions (see diagram 28 on page 53), and the values resulting from such expressions can be assigned to variables. Using this method we can do the following:

```
"fibo.mexe" = main in
{{
main =
  fun()
  {
    rec val fibo =
      fun(n)
      {
        if( n<2 )
        {
          1;
        }
        else
        {
          fibo1(n-1)+fibo1(n-2);
        };
      };
    writeln("fibo(10)="++fibo(10));
```

```
19   };
20 }}
21 *
22 BASIS
23 ;
```

This achieves the same effect as the previous example.

Note that it is possible to have multiple definitions of the same function. For example:

```
1  "fibo.mexe" = main in
2  !
3  {{
4  main =
5    fun()
6    {
7      rec val fibo =
8        fun(n)
9        {
10         if( n<2 )
11         {
12           1;
13         }
14         else
15         {
16           fibo1(n-1)+fibo1(n-2);
17         };
18       };
19     rec fun fibo(n)
20     {
21       if( n<2 )
22       {
23         1;
24       }
25       else
26       {
27         fibo1(n-1)+fibo1(n-2);
28       };
29     };
30     writeln("fibo(10)="++fibo(10));
31   };
32
33 fibo =
34   fun(n)
35   {
36     if( n<2 )
37     {
38       return 1;
```

```
39      }
40      else
41      {
42        return fibo(n-1)+fibo(n-2);
43      };
44    };
45 }}
46 *
47 BASIS
48 ;
```

How then is each call to `fibo` resolved? The general answer is that first a search is made for an inner function definition (i.e. **`rec fun`** `...`) and if such a definition is found in the current scope or some enclosing scope, with the correct number of parameters, then that function definition is used. The innermost definition is used if there are multiple such definitions in enclosing scopes. If no such function definition is found, then a search is made for a variable in the current or enclosing scope that has the name `fibo`, or in the general case the name of the function being called. If such a variable is found then it is assumed that this variable contains a function value (also called a *closure*), and a call is made to the function value. If neither of the above cases apply then it is assumed that the function is a top-level function.

Similar rules apply to unary and binary operations, which can be either inner functions or top-level functions. They can not be variables, however, as there is no syntax to define variables with such names.

Mutual Recursion

Sometimes it is useful to define a collection of mutually recursive values, variable initializations and inner functions. The syntax and semantics of recursive definitions facilitates this. Note that inner function definitions are part of this syntax, see diagram 56 on page 69 for a fuller syntax of such definitions.

Here is an example of two mutually recursive functions. Note that the mutually recursive functions need to be defined in a single declaration.

```
1 "foo.mexe" = main in
2 {{
3 main =
4   fun()
5   {
6     rec
7       fun a(n)
8       {
9         if( n==0 ) {return [];};
10        write("a");
11        b(n-1);
12      },
13      fun b(n)
```

```
14        {
15          if( n==0 ) {return [];};
16          write("b");
17          a(n-1);
18        };
19      a(10);
20    };
21 }}
22 *
23 BASIS
24 ;
```

Here is another example with mutually referencing variables and values.

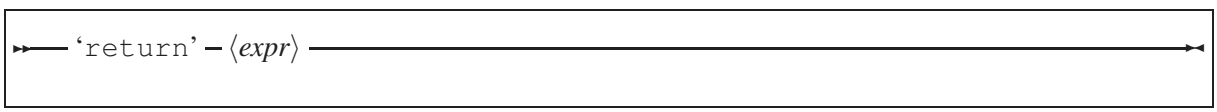
```
1  "foo.mexe" = main in
2  {{
3  main =
4    fun()
5    {
6      rec val a, var b=#[1$a], val a=#[2$b];
7      writeln(streamHead(a));
8      writeln(streamHead(b));
9      writeln(streamHead(streamTail(a)));
10     writeln(streamHead(streamTail(b)));
11   };
12 }}
13 *
14 BASIS
15 ;
```

This will write the integer sequence 2,1,1,2. Note that in order for it to be legal to reference a named value (such as `a` above) or a variable in the right hand side of an initialization, the named value or variable must have been defined before it is referenced. Furthermore, all declared named values must receive an initialization in the same declaration they are declared in.

In a recursive declaration the functions, named values and variables are first all created and then they are initialized in sequence.

2.9.3. The 'return' Expression

The return expression is used to terminate the current function call or method call. Diagram 12 shows the pattern of the return expression. See also diagram 59 on page 70.



Syntax Diagram 12: ⟨*return*⟩

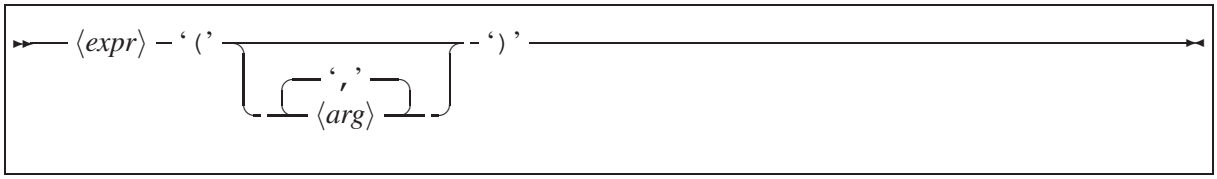
The following program text does the same as the text above, but uses the return statement.

```
"fibo.mmod" =
!
{{
fibo1 =
  fun(n)
  {
    if( n<2 )
    {
      return 1;
    }
    else
    {
      return fibo1(n-1)+fibo1(n-2);
    };
  };

fibo2 =
  fun(n)
  {
    var i = 0, f1 = 1, f2 = 1;
    for(;;)
    {
      if( i==n )
      {
        return f1;
      };
      val nextf = f1+f2;
      i = inc(i);
      f1 = f2;
      f2 = nextf;
    };
  };
}}
;
```

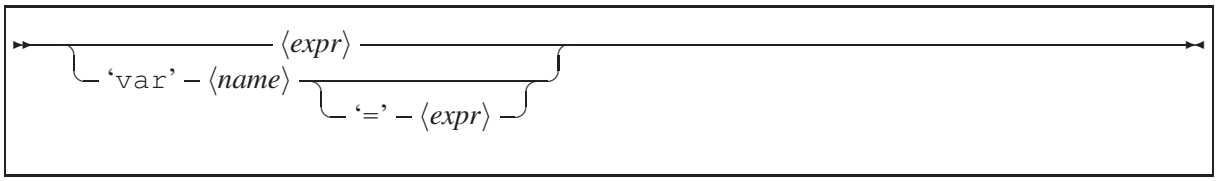
2.10. Function Calls

Diagram 13 on the next page (which is essentially part of diagram 64 on page 71) shows the pattern of a simple function call in Morpho.

The effect of a function call `e(e1,...,eN)` is to evaluate the expressions `e`, `e1`,...,`eN`, in order, and then make a call to the value returned by `e` (assuming `e` is a function, or, in other words, a closure, taking `N` arguments), with the results of `e1`,...,`eN` as arguments. If `e` is a name then the function to be called is resolved as described in section 2.9.2 on page 32, but in



Syntax Diagram 13: ⟨*simple_fun_expr*⟩



Syntax Diagram 14: ⟨*arg*⟩

general `e` can be any expression.

2.10.1. More Complex Function Calls

Morpho supports more complicated compound expressions that also are function calls, but with some syntactic sugar. The purpose is to make it possible for the programmer to define new notation for complicated things such as control structures.

As an example, using this we can for example define our own loop control structure:

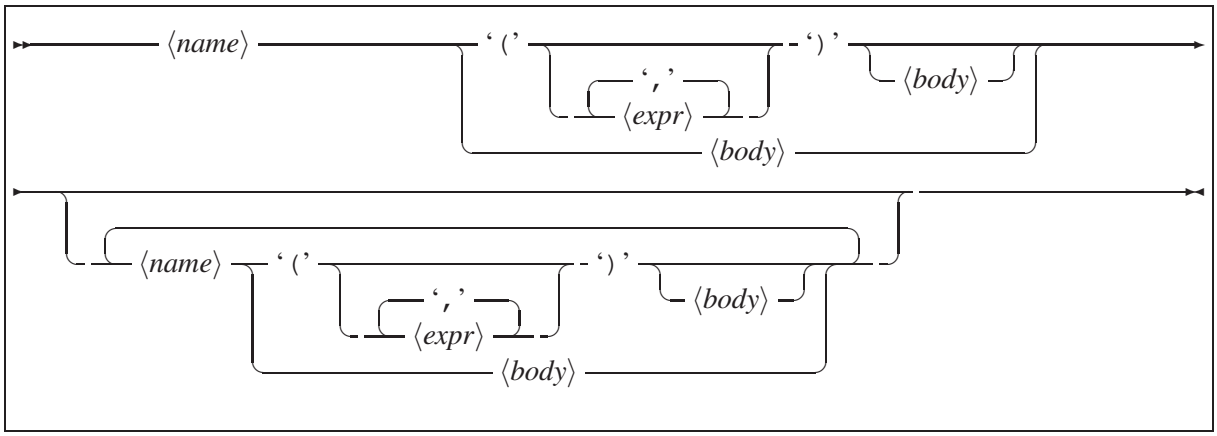
```
rec fun Do_While(@b,@c)
{
        b;
        if( !c ) {return null};
        Do_While(@b,@c);
};

var i=0;
Do
{
        writeln(i);
        i = inc(i);
}
While( @i<10 );
```

This will write the integers 0 to 9. We can also create similar global functions:

```
"test.mexe" = main in
!
{{
main =
  fun()
  {
    var i=0;
    Do
```

```
9       {
10              writeln(i);
11              i = inc(i);
12      }
13      While( @i<10 );
14    };
15
16  Do_While =
17    fun(@b,@c)
18    {
19          b;
20          if( !c ) {return null};
21          Do_While(@b,@c);
22    };
23  }}
24  *
25  BASIS
26  ;
```

The BASIS module contains a few such functions that are intended to be used as control structures, including `forAll_inList_do`, `forAll_inStream_do`, and `sqlForAllRows_fromPreparedQuery_do`.



Syntax Diagram 15: ⟨*compound_fun_expr*⟩

An example of such a call is

```
1  forAll( var x )
2  inList( [1,2,3] )
3  do
4  {
5          writeln(x);
6  }
```

This call is completely equivalent to the call

```
forAll_inList_do( var x, [1,2,3], @{writeln(x);} )
```

In general, a call of the form

```
id1( args1,... )
id2( args2,... )
...
idN( argsN,... )
```

is translated by Morpho into a call

```
id1__id2__idN( args1,...,args2,...,argsN,... )
```

where the number of underscores used to connect the individual identifiers into one identifier is the number of parameters following the identifier in front of the underscores.

For example, the call

```
forAll( var x, var y )
inLists( [1,2,3], ["a","b","c"] )
do
{
        writeln(x++y);
}
```

is translated into

```
forAll__inList__do
  ( var x
  , var y
  , [1,2,3]
  , ["a","b","c"]
  , @{writeln(x++y);}
  )
```

Code bodies are always translated into by-name arguments, and an argument of the form `var x` and `var x=e` is handled by creating a temporary variable whose scope is the rest of the call expression, that is then passed as a by-name argument. So, for example, the call

```
forAll( var x=null, var y )
inLists( [1,2,3], ["a","b","c"] )
do
{
        writeln(x++y);
}
```

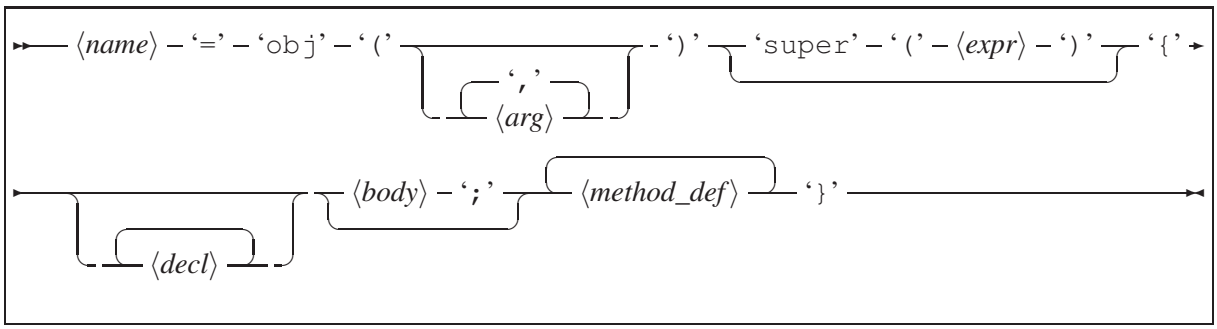
is equivalent to

```
{
  var x=null;
  var y;
```

```
4   forAll__inList__do
5     ( @x
6     , @y
7     , [1,2,3]
8     , ["a","b","c"]
9     , @{writeln(x++y);}
10    )
```

2.11. Object Definitions

Diagram 16 (which includes diagram 51 on page 66) shows the pattern for defining a Morpho object or class. The following example shows a couple of object definitions in a module.



Syntax Diagram 16: ⟨*obj_def*⟩

```
1  "objexample.mmod" =
2  !
3  {{
4  stack =
5    obj(max)
6    {
7      var arr = makeArray(max);
8      val limit = max;
9      var pos = 0;
10     msg push(x)
11     {
12       pos<limit ||
13         (throw "Attempt␣to␣push␣on␣a␣full␣stack");
14       arr[pos] = x;
15       pos = inc(pos);
16     };
17     msg pop()
18     {
19       pos>0 ||
20         (throw "Attempt␣to␣pop␣from␣an␣empty␣stack");
21       pos = dec(pos);
```

```
22        return arr[pos];
23      };
24      msg isEmpty()
25      {
26        return (pos==0);
27      };
28      msg isFull()
29      {
30        return (pos==limit);
31      };
32    };
33
34  extendedStackOfDouble =
35    obj() super stack()
36    {
37      var count = 0.0;
38      var sumx = 0.0;
39      var sumxx = 0.0;
40      msg push(x)
41      {
42        sumx = sumx + x;
43        sumxx = sumxx + x*x;
44        count = count + 1.0;
45        super.push(x);
46      };
47      msg pop()
48      {
49        val x = super.pop();
50        sumx = sumx - x;
51        sumxx = sumxx - x*x;
52        count = count - 1.0;
53        return x;
54      };
55      msg count()
56      {
57        return count;
58      };
59      msg average()
60      {
61        return sumx/count;
62      };
63      msg stdDev()
64      {
65        return
66          "java.lang.Math".##sqrt(
67             (sumxx-sumx*sumx/count)/count
```

```
68            );
69        };
70    };
71 }}
72 ;
```

2.11.1. ‘this’ And ‘super’

There are two special expressions in Morpho that can only be used inside methods of objects. The expression **`this`** is used to refer to the current object. For example, inside a method, the expression **`this`**`.f(1)` is used to send the message `f` with the argument `1` to the current object. The method to be executed is determined by traversing the inheritance of the object, from the bottom to the top (the topmost ‘superclass’) and executing first method found that is associated with the message.

Sometimes, however, there is the need to send a message to the current object and make sure that the corresponding method to be executed is a method of a ‘superclass’ of the object rather than executing the first method in the inheritance path. In that case we use the special expression **`super`**, which is also only available inside methods. We see examples of this above in the methods for `push` and `pop` in the subclass `extendedStackOfDouble`.

2.12. Constructing Objects

An object is constructed by calling a constructor. Seen from the outside, a constructor is simply a function. The syntax of defining a constructor is different, however. Here are a couple of definitions, one is a constructor, the other one is a function.

```
1  "example.mmod" =
2  {{
3  f =
4    fun()
5    {
6      1;
7    };
8
9  g =
10   obj()
11   {
12     msg h()
13     {
14       2;
15     };
16   };
17 }}
18 ;
```

Seen from the outside both `f` and `g` are functions taking no arguments and returning some value. In the case of `f`, the value returned is the integer `1`, whereas a call `g()` returns an object which responds to the message `h()` by returning the integer `2`.

2.13. Scope of Declarations

We have seen various types of declarations, but they can be split into two major types; recursive declarations that start with the keyword 'rec' and non-recursive declarations that do not. Both these types of declarations can involve initializations, especially the recursive ones.

The scope of a name defined in a recursive declaration extends from the *beginning* of the declaration (i.e. just after the keyword 'rec' that starts the declaration) and to the end of the block that contains the declaration, i.e. up to the closing curly brace, '}'.

On the other hand the scope of a name defined in a non-recursive declaration extends from the *end* of the declaration and up to the end of the containing block.

Take for example the following program that demonstrates the difference between a recursive declaration and a non-recursive one in a simple way.

```
"example.mexe" = main in
{{
main =
  fun()
  {
    val x=#[1];
    {
      val x=#[2$x];         ;;; x is #[2,1]
      writeln(streamHead(streamTail(x)));
    };
    {
      rec val x=#[2$x];     ;;; x is #[2,2,2,2,...]
      writeln(streamHead(streamTail(x)));
    };
  };
}}
*
BASIS
;
```

This program will write the two integers 1, 2.

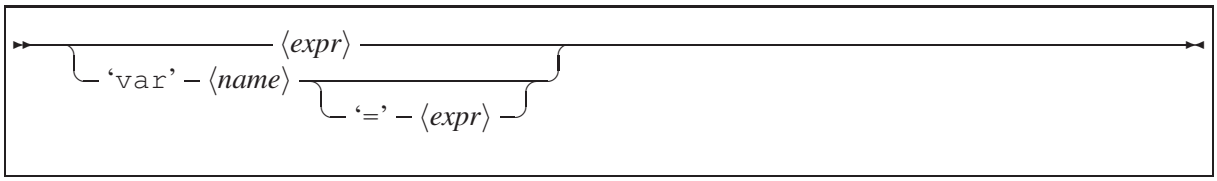
2.14. Method-Call Expressions

There are various types of expressions in Morpho for manipulating both Java objects and Morpho objects.

2.14.1. Morpho Method Invokations

Morpho also has it's own objects as well as supporting the manipulation of Java objects. The syntax for method invocations on Morpho objects is shown in diagram 18.

In order to define Morpho method invocations we need first to define the syntax for arguments in diagram 17 As the diagram shows, an argument is either an expression or a vari-



Syntax Diagram 17: ⟨*arg*⟩

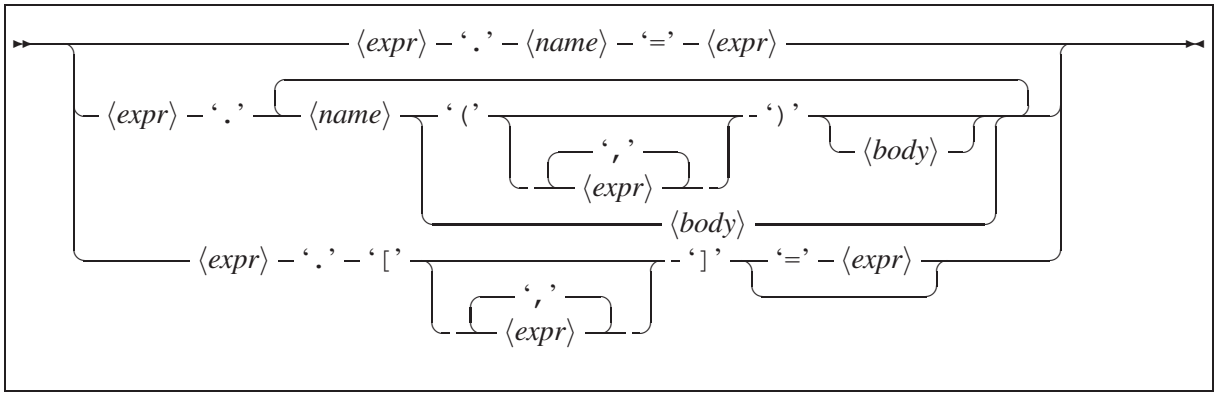
able declaration. The effect of a variable declaration as an argument is to create a temporary variable whose scope is the rest of the invocation. That variable is then sent as a by-name argument, which is equivalent to saying that a thunk referring to the variable is created and the thunk is sent as a by-value argument.

For example the invocation

```
1 x.f(var y=1)
```

is equivalent to

```
1 {var y=1; x.f(@y)}
```



Syntax Diagram 18: ⟨*morpho_method_invocation*⟩

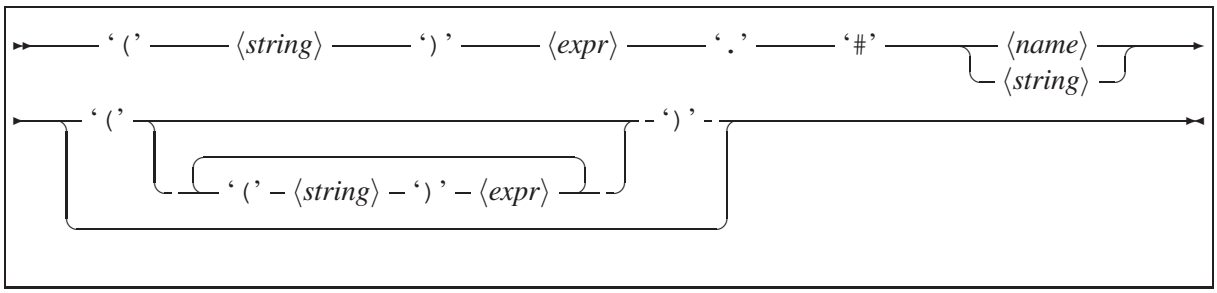
There are five patterns of Morpho method invocations all of which can be seen in diagram 18.

- The effect of a regular Morpho method invocation such as `e.f(e1,...eN)` is to first evaluate the expression `e`, then evaluate the expressions `e1`,...,`eN` in sequence, and finally send the message `f` to the result from `e` with the results from `e1`,...,`eN` as arguments. The result from that invocation is then the result from evaluating the total expression.

- The effect of an accessor method invocation such as `e.x` is to evaluate `e` and send the accessor message `x` to the result. The result from the method that the message invokes becomes the result of the whole expression.
- The effect of an accessor method invocation such as `e1.x=e2` is to evaluate `e1` and `e2`, in that order, and send the accessor message `x=` to the result from `e1` with the result from `e2` as argument. The result from the method that the message invokes becomes the result of the whole expression.
- The effect of a array accessor Morpho method invocation such as `e.[e1,...eN]` is to first evaluate the expression `e`, then evaluate the expressions `e1`,...,`eN` in sequence, and finally send the array accessor message to the result from `e` with the results from `e1`,...,`eN` as arguments. The result from that invocation is then the result from evaluating the total expression.
- The effect of a array assignment Morpho method invocation such as `e0.[e1,...eN]=e` is to first evaluate the expression `e0`, then evaluate the expressions `e1`,...,`eN` and `e` in sequence, and finally send the array assignment message to the result from `e0` with the results from `e1`,...,`eN` and `e` as arguments. The result from that invocation is then the result from evaluating the total expression.

2.15. Java Method-Call Expressions

Diagram 19 shows the syntax to invoke a Java instance method in Morpho.



Syntax Diagram 19: ⟨*java_instance_method_invocation*⟩

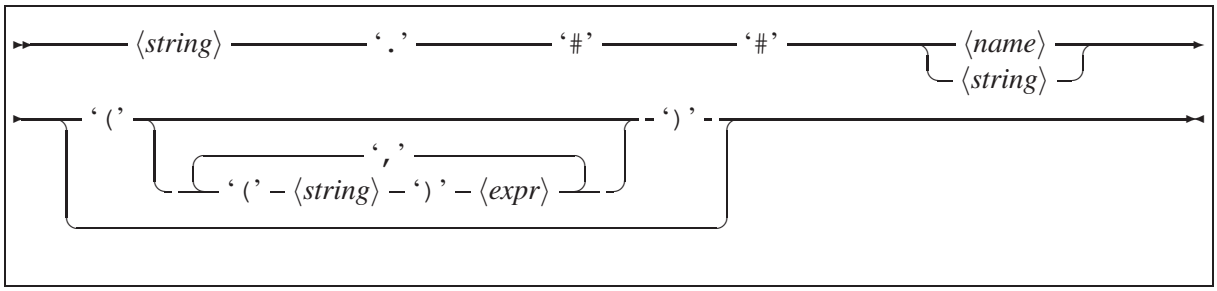
The following example shows how to extract a substring using the substring method of class java.lang.String.

```
1 var str,sub;
2 str = "Hello World";
3 sub = ("java.lang.String")str.#substring(("int")6,("int")11);
4 writeln(sub);
```

Executing this code will cause the word **World** to be written. The parenthesized string that the expression starts with should be the fully qualified class name for the object that the target expression returns. In the example the target expression is a Java string so the fully qualified

class name is java.lang.String. The name of the method to invoke should follow the '#' character, either as an identifier or as a string. Allowing a string there makes it possible to invoke methods that have the same name as a Morpho keyword. The argument list is a sequence of pairs where we need to both specify a type and a value. The type is the parenthesized string in front of the corresponding value. The type is either a fully qualified class name or one of the Java built-in primitive types, i.e. byte, short, int, long, float, double, char or boolean.

Diagram 20 shows the syntax to invoke a Java class method in Morpho.



Syntax Diagram 20: ⟨*java_class_method_invocation*⟩

An example is the following

```
1 val time = "java.lang.System".##nanoTime();
```

This will cause a call to the class method nanoTime in the class java.lang.System. When arguments are used in a class method call then the types of the arguments must be specified just as in Java instance method calls above.

An expression `(t)e.#f((t1)e1,...,(tN)eN)` has the effect of first evaluating `e` and then evaluating `e1`,...,`eN`, in order, and then sending the Java message `f` to the result from `e`, with the results from `e1`,...,`eN` as arguments. The expressions `t`,`t1`,...,`tN` must be string literals and must be fully qualified class names or primitive Java type names. The message `f` is resolved at run-time on the basis of its name, the number and types of the arguments `t1`,...,`tN`, and the type of the receiver of the message, `t`.

For example, imagine we have Java classes A and B where B is a subclass of A, as follows:

```
1  class A
2  {
3    A f( A x )
4    {
5      return x;
6    }
7  }
8
9  class B extends A
10 {
11   A f( A x )
12   {
13     return x;
14   }
```

```
15 }
```

Now assume that the Morpo variable x contains a reference to a Java object of class B. Then the invocation `("B")x.#f(("B")x)` will fail because the run-time system will try to find a method taking an argument of type B, which it won't find. In order to invoke the method you should specify the precise type of argument as declared in the class. The invocation `("A")x.#f(("A")x)` will succeed because it's effect is to look for a method named `f` which takes a single argument of type A and send `x` as that argument (`x` is, of course, both of type A and of type B because B is a subclass of A).

The following Morpho code is another example.

```
1 y = ("java.math.BigDecimal")y.#add(("java.math.BigDecimal")y);
```

Here the programmer specifies that `y` is of type java.math.BigDecimal and uses the Java instance method add to double its value.

Java class methods can be invoked in Morpho using expressions such as `t.##f((t1)e1,...,(tN)eN)` (note the double '#', as opposed to the single '#' used in instance method invocations). The expression `t` must be a string literal and must be the full name of a Java class. The same holds for `t1`,...,`tN`. `f` may be either a name or a string and should be the name of a static method in the class. The static class method denoted by `f` is invoked for the class denoted by `t`, with the arguments being the values yielded by `e1`,...,`eN`, which must be of types denoted by `t1`,...,`tN`. The order of evaluation of the expressions `e1`,...,`eN` is, as you would expect, from left to right.

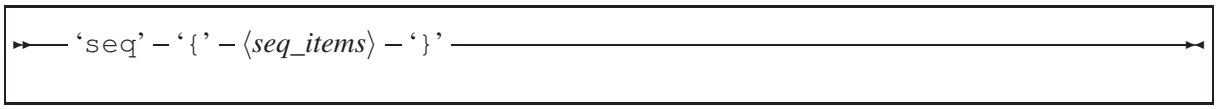
The following is an example of using a call to a static class method to compute pi.

```
1 val PI = "java.lang.Math".##atan2(("double")0.0,("double")-1.0);
```

2.16. The 'seq' Expression

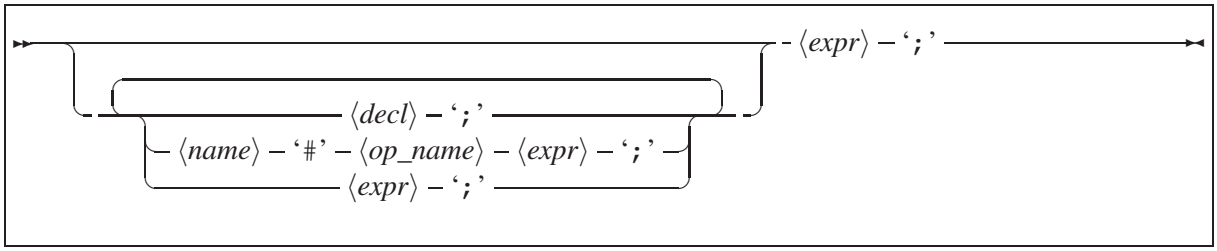
The purpose of the seq expression is to support a well-known style of functional programming where values from a container are fed through a kind of production line of functions where each function manipulates a value and produces values into another container. The seq expression is closely related to do expressions and list comprehensions in the functional programming language Haskell.

Syntax diagrams 21 (also part of 65 on page 72) and 22 on the next page (same as 70 on page 74) show the pattern of the 'seq' expression in Morpho. The seq expres-



Syntax Diagram 21: ⟨*seq_expr*⟩

sions are really just syntactic sugar, i.e. they are a little more convenient way of expressing things that can already be expressed in Morpho in other ways. Consequently, the semantics of a seq expression can be described by showing alternate ways of writing a semantically equivalent expression.



Syntax Diagram 22: ⟨*seq_items*⟩

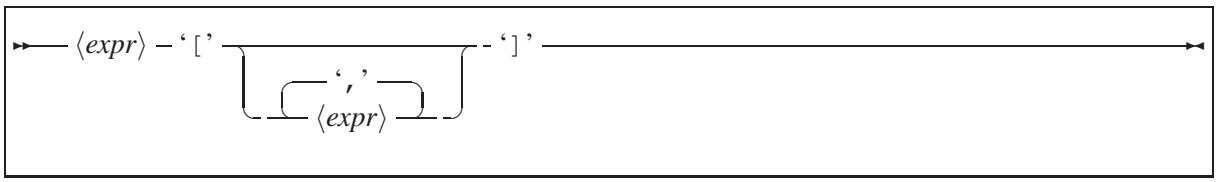
- A seq expression **`seq`** `{ expr; }` where `expr` is an expression is simply equivalent to the expression `expr`.
- A seq expression **`seq`** `{ decl; ... }` where decl is a declaration is equivalent to the expression `{ decl; seq {...}; }`.
- And finally, and most important, a seq expression **`seq`** `{ x #<<= e; ... }` is equivalent to the binary function (operator) call `e >>= fun(x){...}`. The operator name `<<=` can be any binary operator name, the corresponding operator name `>>=` in the semantically equivalent expression is always derived by simply changing all '>' characters to '<' and changing all '<' characters to '>', i.e. by reversing the direction of those characters.

2.17. Array Expressions

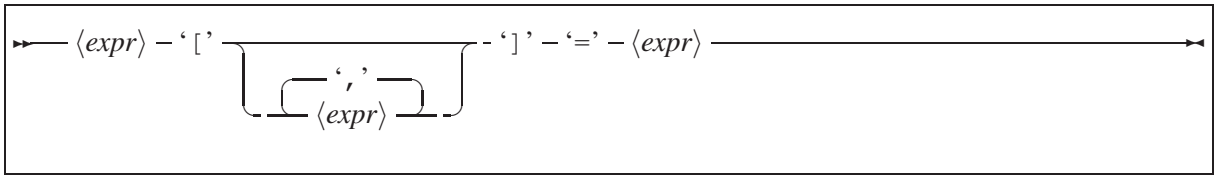
Morpho supports both object-oriented and non-object-oriented arrays. Fetching a value from a non-object-oriented array or inserting a value into it is achieved by calling some function. In the case of object-oriented arrays the same things are achieved by sending messages to the array.

2.17.1. Non-Object-Oriented Arrays

Syntax diagrams 23 and 24 on the following page show the patterns of Morpho expressions used to fetch values from and store values into arrays. Note that these two diagrams, although they are syntactically correct, ignore the effects of the different precedences of the diverse syntactic constructs in Morpho. Diagram 64 on page 71 and related diagrams are more appropriate in that regard.



Syntax Diagram 23: ⟨*array_get_usage*⟩



Syntax Diagram 24: ⟨*array_set_usage*⟩

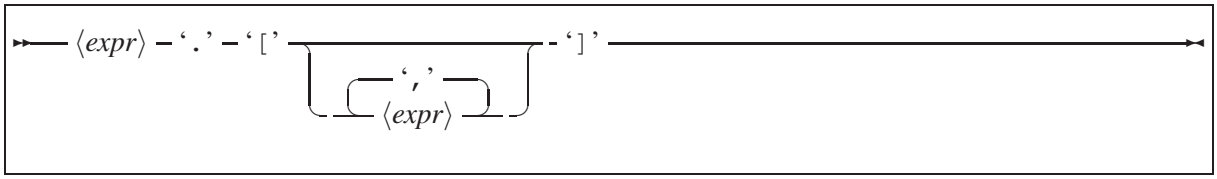
The following module defines simple-minded non-object-array of two position denoted by indexes `0` and `1`.

```
"simplearray.mmod" =
{{
makeSimpleArray =
  fun()
  {
    []:[];
  };

get[] =
  fun(a,i)
  {
    i==0 && (return head(a));
    return tail(a);
  };

set[] =
  fun(a,i,x)
  {
    if( i==0 )
    {
      setHead(a,x);
      return x;
    };
    setTail(a,x);
    x;
  };
}}
;
```

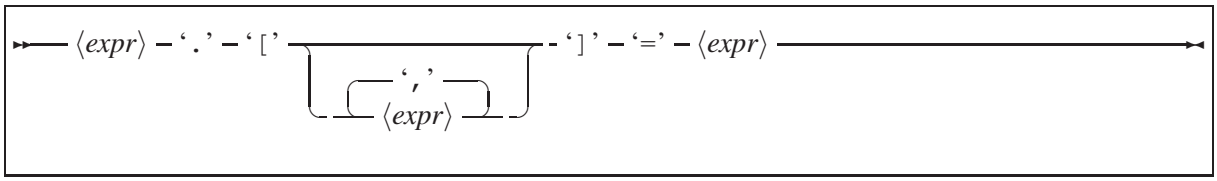
The two exported functions `get[]` and `set[]` are semantically just like any other Morpho functions. Syntactically, however, their names make them special in that they are called using expressions `a[i]` and `a[i]=e`, respectively, where `i` and `e` are any expressions. Similar holds for arrays of more than one dimension or fewer than one (i.e. zero) dimensions. A function with signature `fun(a,i1,...,iN)` (i.e. a function taking N+1 arguments) can be used as a `get[]` function for N-dimensional arrays, and a function with signature `fun(a,i1,...,iN,x)` can be used as a `set[]` function. It is the programmers responsibility to make sure that these behave like array get and set functions.

2.17.2. Object-Oriented Arrays

Syntax diagrams 25 and 26 show the patterns of Morpho expressions used to fetch values from and store values into object-oriented arrays. Note that, as with the non-object-oriented arrays, these two diagrams, although they are syntactically correct, ignore the effects of the different precedences of the diverse syntactic constructs in Morpho. Again, diagram 64 on page 71 and related diagrams are more appropriate in that regard.



Syntax Diagram 25: ⟨*obj_array_get_usage*⟩



Syntax Diagram 26: ⟨*obj_array_set_usage*⟩

The following module defines simple-minded object-oriented array of two position denoted by indexes `0` and `1`.

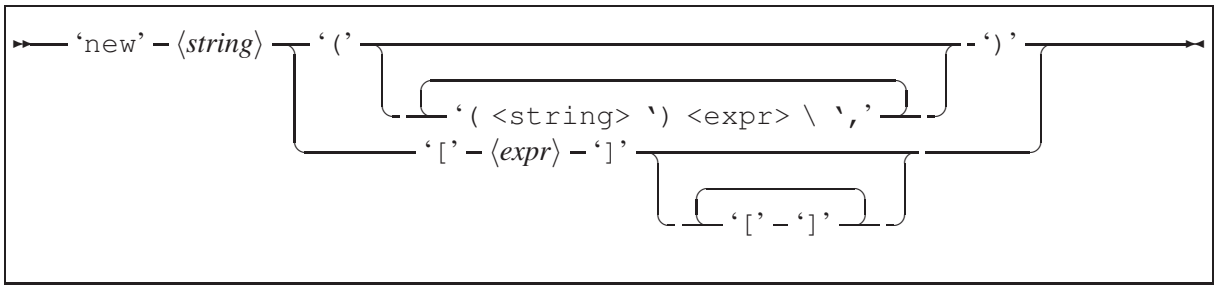
```
"simpleobjarray.mmod" =
{{
makeSimpleObjArray =
  obj()
  {
    var pos0, pos1;
    msg get[i]
    {
      i==0 && (return pos0);
      return pos1;
    };
    msg set[i]=x
    {
      if( i==0 )
      {
        pos0 = x;
        return x;
      };
      pos1 = x;
      x;
    };
  };
```

```
23 }}
24 ;
```

The two messages `get[]` and `set[]` are semantically just like any other Morpho messages. Syntactically, however, their names make them special in that they are called using expressions `a.[i]` and `a.[i]=e`, respectively, where `i` and `e` are any expressions. Similar holds for arrays of more than one dimension or fewer than one (i.e. zero) dimensions. A message with signature **`msg`**`(i1,...,iN)` (i.e. a message taking N arguments) can be used as a `get[]` message for N-dimensional arrays, and a message with signature **`msg`**`(i1,...,iN,x)` can be used as a `set[]` message. It is the programmers responsibility to make sure that these behave like array get and set messages.

2.18. Java Construction Expressions

Morpho has a special expression to directly construct a Java object using a Java class constructor. Diagram 27 shows the pattern of the Java object construction expression. The effect



Syntax Diagram 27: ⟨*java_construction_expr*⟩

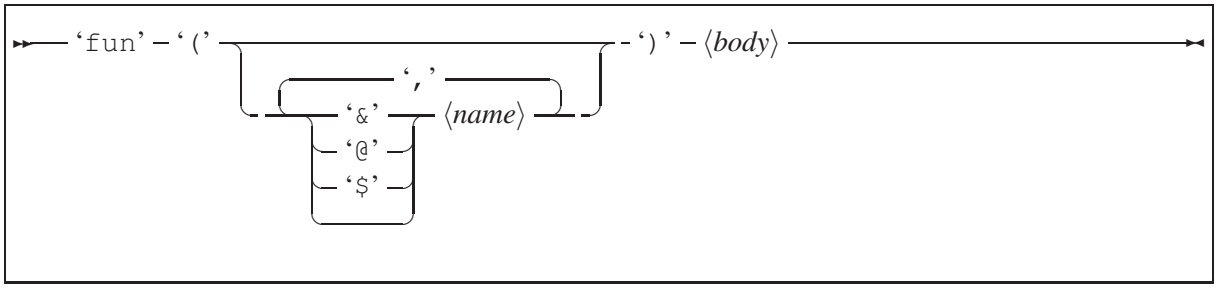
of an expression lstinlinelnew "A"(("B1")e1,...,("Bn")en)| is to call a Java class constructor for class `A` with arguments `e1,...,en`. The constructor called is found by searching for a constructor that takes n arguments of types `B1,...,Bn`. If no such constructor exists or if the values returned by the expressions `e1,...,en` are not of types `B1,...,Bn` then a run-time exception will occur. If no error occurs the value of the expression will be a reference to the newly created object of type `A`.

2.19. Delayed-Evaluation Expressions

Morpho has expressions that delay execution of contained expressions. The simplest such expression is the function expression.

2.19.1. Function Expression

Diagram 28 on the following page shows the pattern of the anonymous function expression. Such function values can be assigned to variables, returned as values from functions, and in



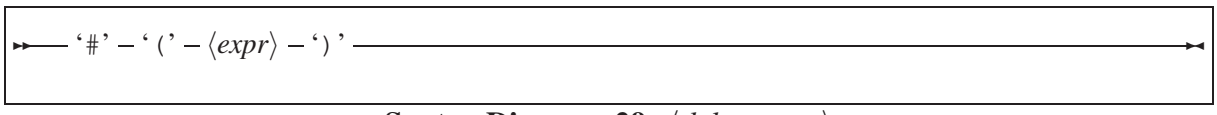
Syntax Diagram 28: ⟨*anonymous_fun_def*⟩

general sent anywhere, just like any other Morpho values. The following program code shows examples.

```
"funexample.mexe" = main in
{{
main =
  fun()
  {
    var f,g,x;
    f = fun(z){z+x;};
    x = 10;
    writeln(f(1));              ;;; Writes 11
    g = f;
    x = 100;
    writeln(g(1));              ;;; Writes 101
    x = 1000;
    writeln(fun(z){z+x;}(1));   ;;; Writes 1001
  };
}}
*
BASIS
;
```

2.19.2. Memoized Delay

Diagram 29 shows the pattern of the delay expression. Evaluating a delay expression does



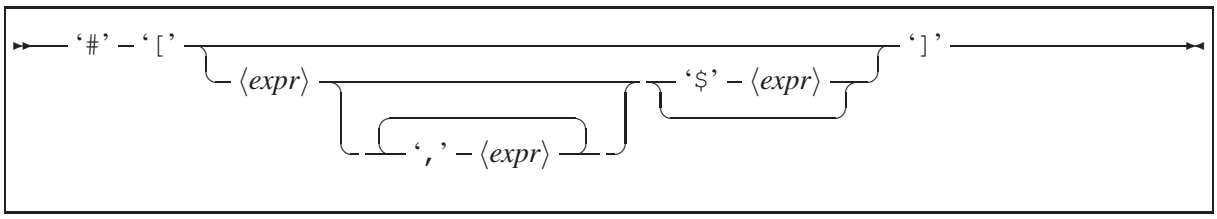
Syntax Diagram 29: ⟨*delay_expr*⟩

not cause the enclosed expression to be evaluated but instead creates a *promise* that can later be evaluated using the `force` function in the BASIS module. The two expressions `e` and `force($(e))` are semantically equivalent for any expression `e`. However, forcing a promise a second time (or more times) does not cause subsequent evaluations but rather causes the

result of the first evaluation to be returned each time. If multiple Morpho tasks attempt to force the same promise only one task will be allowed to start evaluating the subexpression. The other tasks will be blocked until the first evaluation succeeds and will then receive the result from that first evaluation. Because a promise remembers the result of its first evaluation this is sometimes called *memoization*. We say that the result of the computation is *memoized*.

2.19.3. Stream Expression

Stream expressions are used to construct sequences where the values in the sequence are generated on demand. Diagram 30 shows the pattern of stream expressions. Stream expressions,



Syntax Diagram 30: ⟨*stream_expr*⟩

like the delayed evaluation expressions above, are really just syntactic sugar, i.e. they are more convenient ways of expressing things that can already be expressed in other ways in Morpho.

- A stream expression `#[]` is equivalent to the expressions `[]` and **`null`**.
- A stream expression `#[e]`, where `e` is a single expression, is equivalent to `[e\$\$([])]`. Please note two consequences of the embedded delay expression.
  - The computation of the tail is memoized (see above).
  - If multiple tasks attempt evaluation of the tail of a stream only one of the tasks will be allowed to perform the evaluation. The other tasks will be blocked until the result is ready.
- A stream expression `#[e1,...]` is equivalent to `[e1 \$ \$(#[...])]`.
- A stream expression `#[e1 \$ e2]` is equivalent to `[e1 \$ \$(e2)]`, which is equivalent to `e1:\$(e2)`.

There are also two BASIS functions, `streamHead` and `streamTail` for deconstructing streams. `streamHead` returns the head of a stream while `streamTail` returns its tail.

The following relations hold.

- `streamHead(#[e1 \$ e2])` returns the same value as `e1`.
- `streamTail(#[e1 \$ e2])` returns the same value as `e2`.

Note the similarity of these stream relations to the relations that hold for lists, i.e.

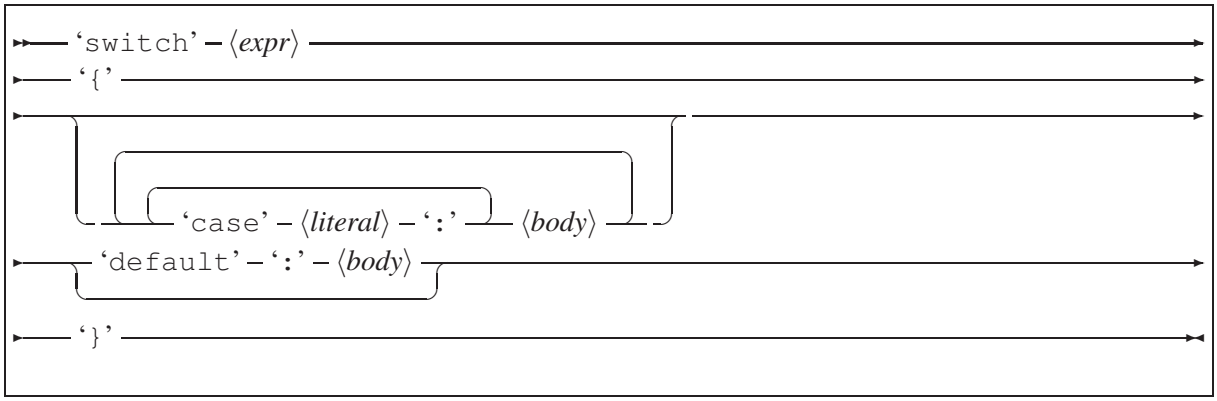
- `head([e1 \$ e2])` returns the same value as `e1`.

- `tail([e1 \$ e2])` returns the same value as `e2`.

The difference is in the way the tail is evaluated. In the case of lists, when the expression `[e1 \$ e2]` is evaluated, the subexpression `e2` is immediately evaluated. In the case of streams, on the other hand, when the expression `#[e1 \$ e2]` is evaluated, the subexpression `e2` is *not* immediately evaluated. The expression `e2` will be evaluated the first time the function `streamTail` is applied to the result of evaluating `#[e1 \$ e2]`. Note also that for any value `x` the expressions `streamTail(x)` and `force(tail(x))` are equivalent, assuming that the functions applied are the ones in the BASIS module.

2.20. The 'switch' Expression

Diagram 31 (same as diagram 73 on page 74) shows the pattern of the switch expression. The effect of the switch expression is to evaluate <*expr*> and then search in some order

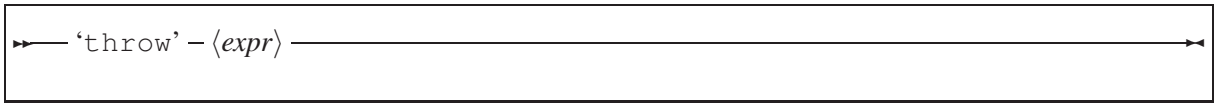


Syntax Diagram 31: ⟨*switch_expr1*⟩

through the cases and look for a literal that returns the same value as the expression. If such a literal is found then the corresponding body is evaluated and the resulting value becomes the value of the switch expression. If no such literal can be found then the default body is evaluated, if present, and the resulting value becomes the value of the switch expression. If no matching literal can be found and no default is present then the switch expression returns some unspecified value. Those familiar with the switch expression in C and C++ should note that there is no fall-through in the Morpho switch expression as in C and C++. Neither is there any relationship between a switch expression and a break expression.

2.21. Exception Handling Expressions

Morpho supports exceptions in a fashion similar to Java, C# and C++. Any value can be thrown as an exception, as in C++, but contrary to the way Java and C# support exceptions.



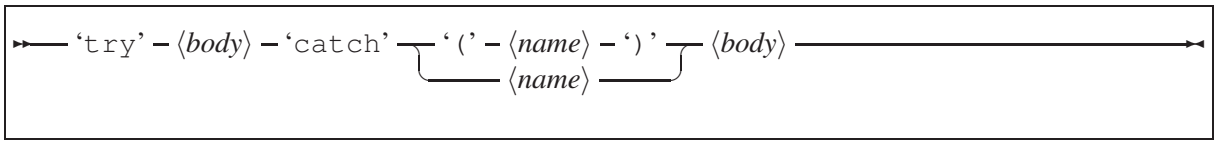
Syntax Diagram 32: ⟨*throw-expr*⟩

2.21.1. The 'throw' Expression

Diagram 32 shows the pattern of the throw expression. The effect of a throw expression such as **`throw`** `e` where e is any Morpho expression, is to evaluate `e`, and throw the resulting value as an exception. This means abandoning the current computation in the current fiber in the current thread and transferring control to the catch part of the most recently entered (and not exited) try part of a try-catch expression (see below).

2.21.2. The 'try-catch' Expression

Diagram 33 shows the pattern of the try-catch expression. The effect of a try-



Syntax Diagram 33: ⟨*try-catch-expr*⟩

catch expression such as **`try`**`{t1;...}`**`catch`**`(e){c1;...}` or the equivalent expression **`try`**`{t1;...}`**`catch`** `e {c1;...}` is to evaluate the try body `{t1;...}` and if an expression is thrown during that evaluation (and not cought by another try-catch) then the catch body `{c1;...}` is evaluated with the variable `e` being initialized with the value of the exception, i.e. the value resulting from the evaluation of the expression that is the argument of the throw expression (see above).

3. Syntax

The Morpho syntax is reminiscent of C, C++, C# and Java.

3.1. Elements Of The Language

3.1.1. Keywords

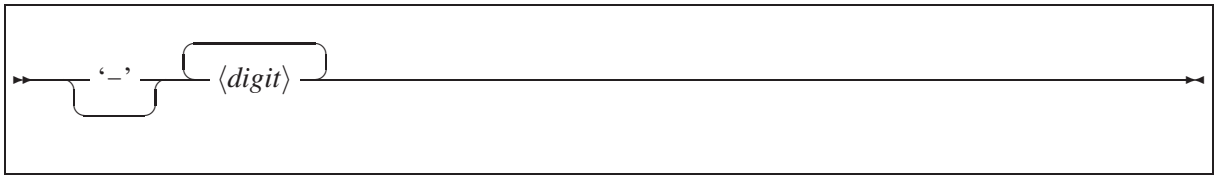
Morpho has the following keywords:

> 'break' 'case' 'catch' 'continue' 'default' 'else' 'elsif' 'false' 'final' 'for' 'fun' 'globalvar' 'if' 'in' 'keysymbol' 'machinevar' 'msg' 'new' 'null' 'obj' 'taskvar' 'rec' 'return' 'seq' 'super' 'switch' 'taskvar' 'this' 'throw' 'true' 'try' 'val' 'var' 'while'

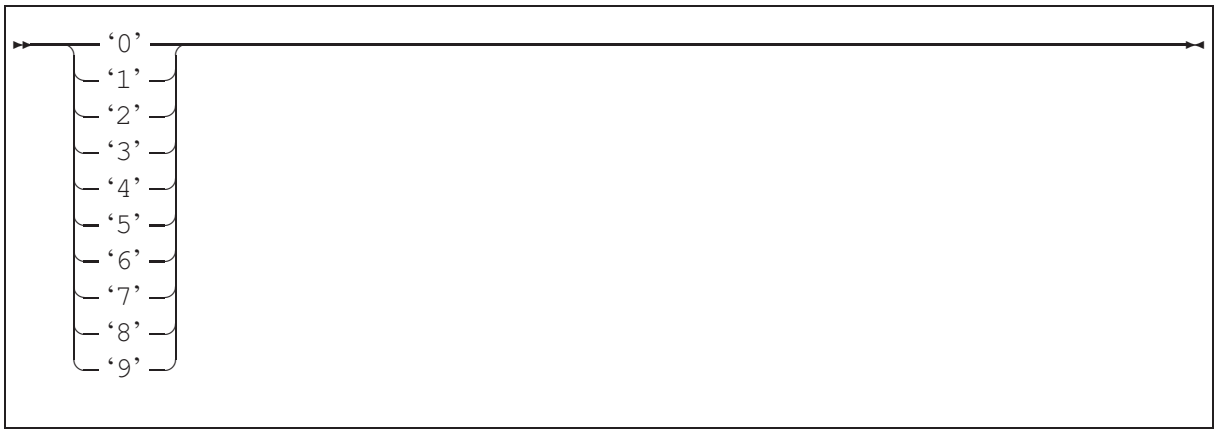
3.1.2. More language elements

There are several kinds of literals and language elements in Morpho.

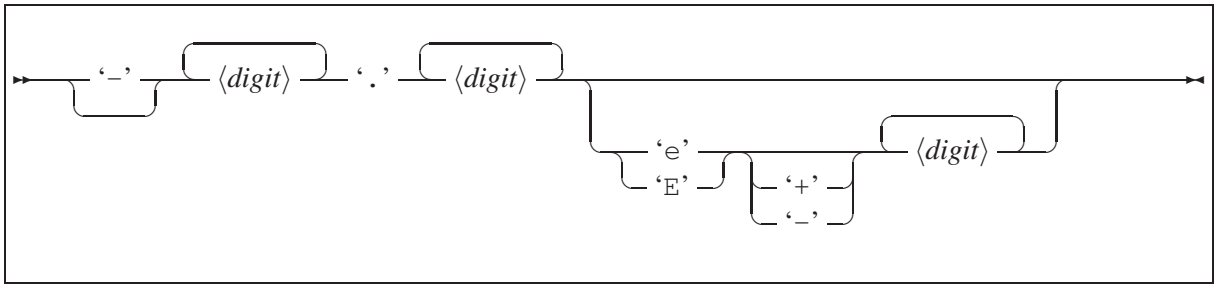
- Integer literals are described by syntax diagram 34.
- Floating point (double) literals are described by syntax diagram 36 on the following page.
- Character literals are described by syntax diagram 37 on the next page.
- String literals are described by syntax diagram 38 on page 59.
- Literals in general are described by syntax diagram 40 on page 59.
- Names are described by syntax diagram 41 on page 60.
- Operation names are described by syntax diagram 43 on page 60.



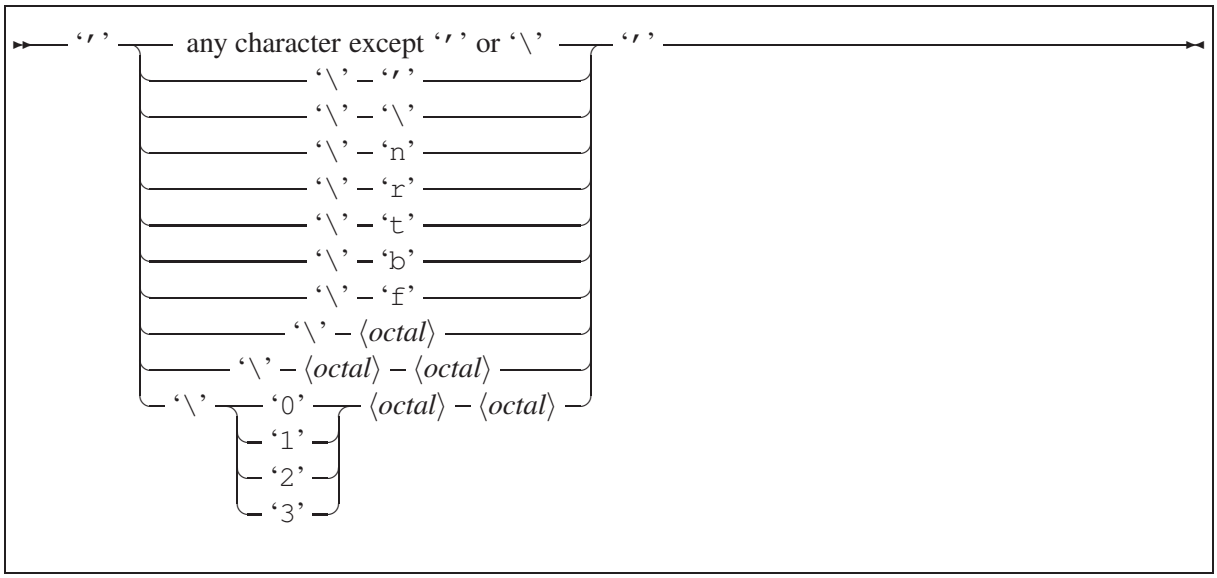
Syntax Diagram 34: ⟨*integer*⟩



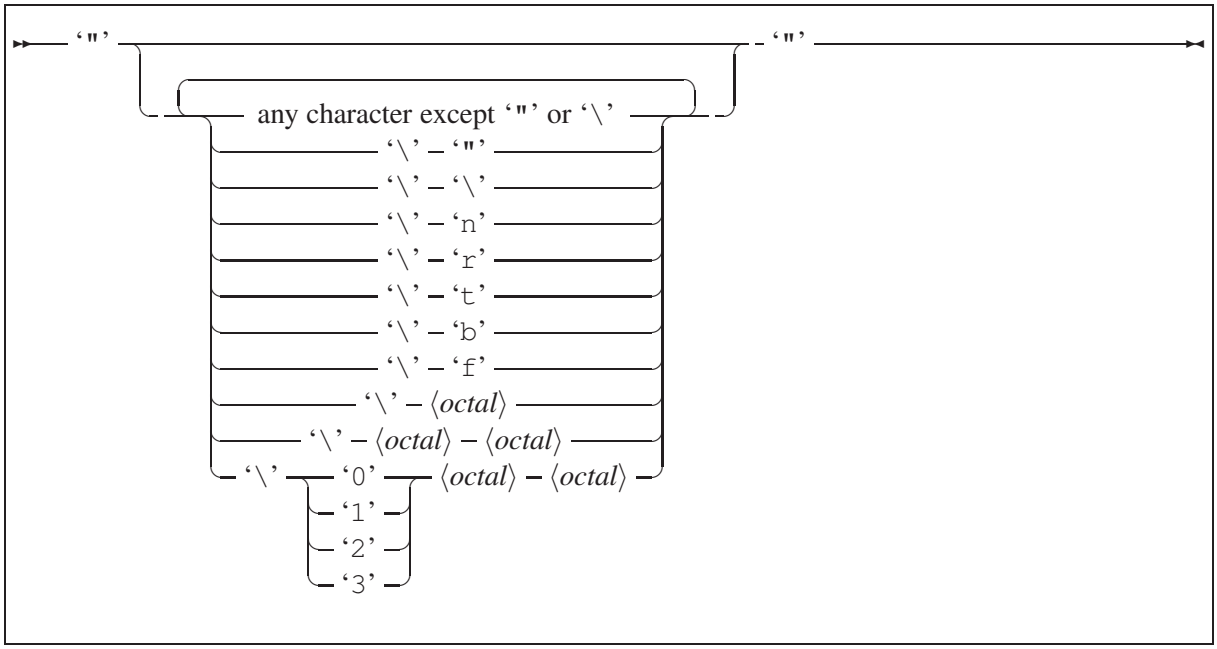
Syntax Diagram 35: ⟨*digit*⟩



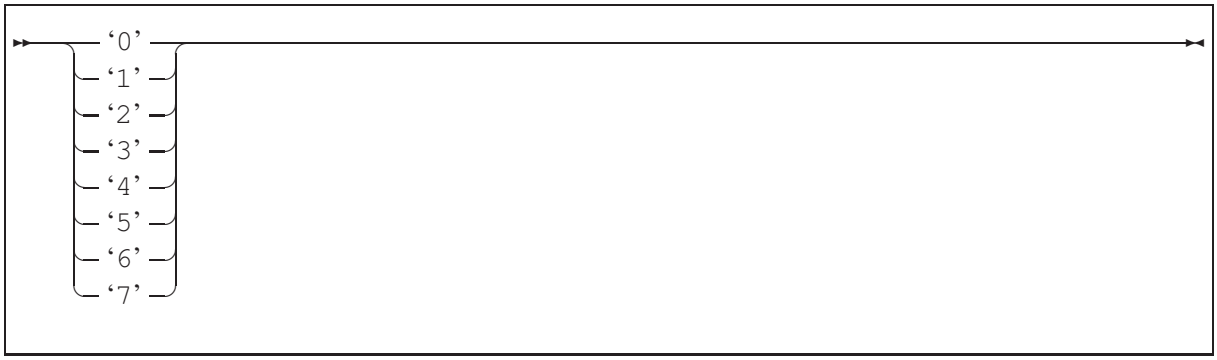
Syntax Diagram 36: ⟨*double*⟩



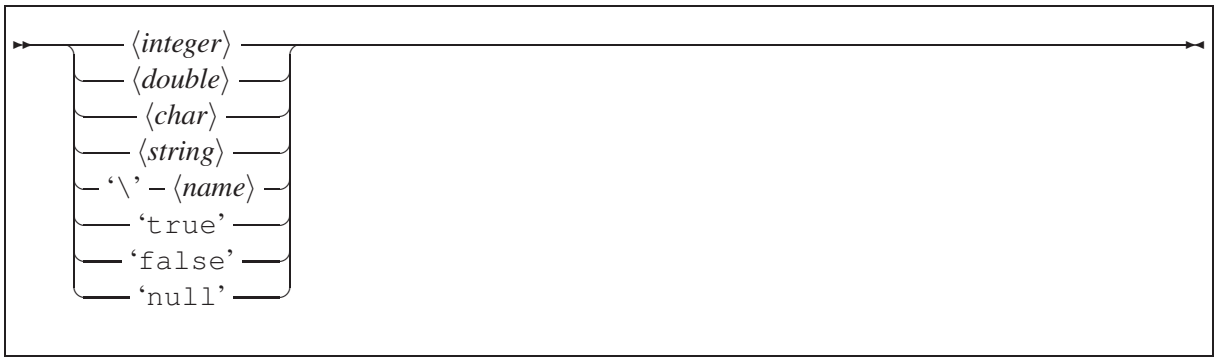
Syntax Diagram 37: ⟨*char*⟩



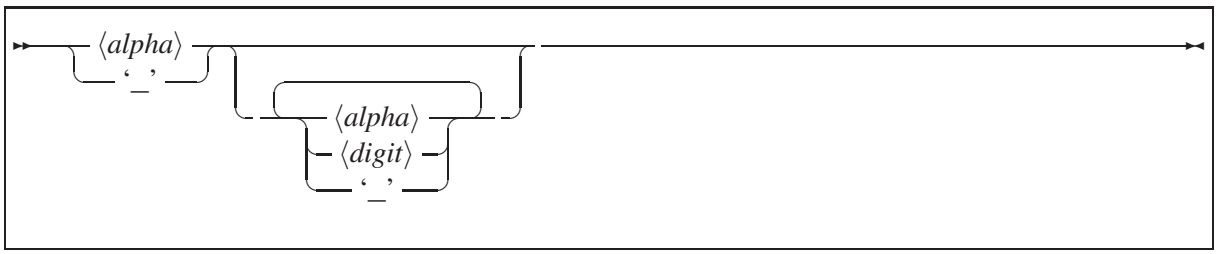
Syntax Diagram 38: ⟨*string*⟩



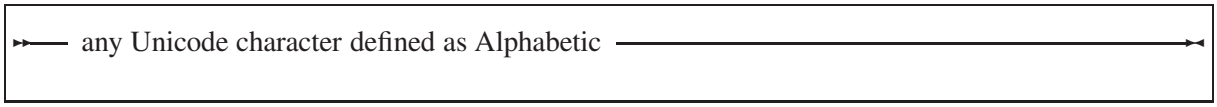
Syntax Diagram 39: ⟨*octal*⟩



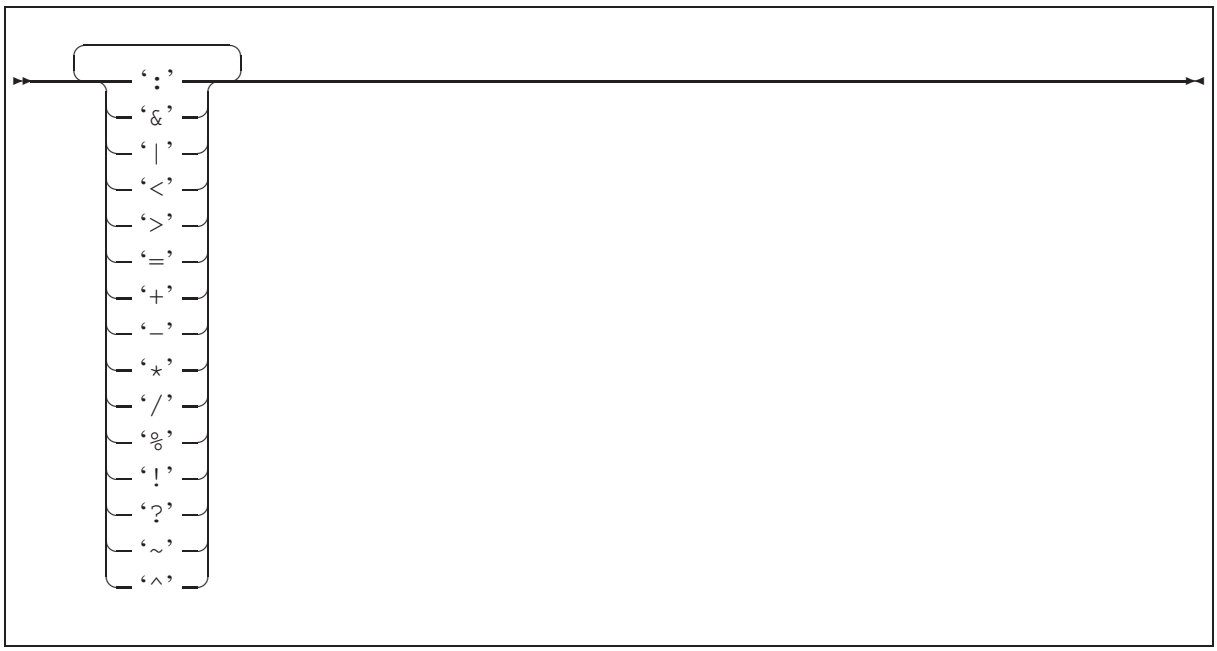
Syntax Diagram 40: ⟨*literal*⟩



Syntax Diagram 41: ⟨*name*⟩



Syntax Diagram 42: ⟨*alpha*⟩



Syntax Diagram 43: ⟨*opname*⟩

3.1.3. Special Symbols

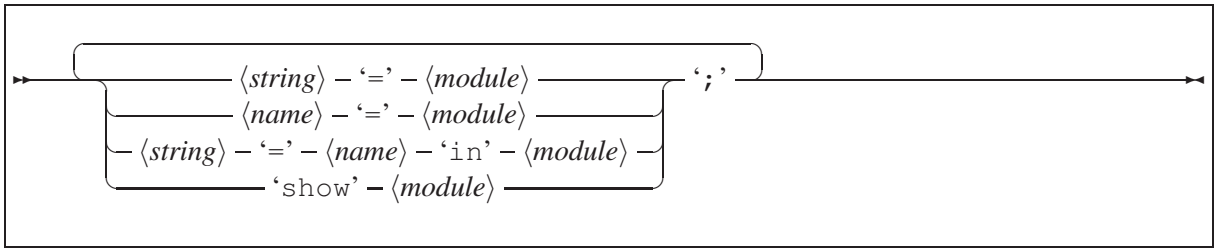
Morpho has the following special symbols: ’(’, ’)’, ’{’, ’}’, ’[’, ’]’, ’$’, ’#’, ’@’, ’,’, ’;’, ’.’. In addition to each of these symbols having their special meaning, the special symbols act as delimiters between other elements of the language as does white space (space, newlines and tabbing characters).

3.1.4. Comments

Comments can be anywhere that white space is allowed, i.e. between any lexical elements (elements of the language). There are two types of comments, line comments and multiline comments. Line comments start with ’;;;’ and extend to the end of the line. Multiline comments start with ’{;;;’ and extend to a closing ’;;;}’. Nested multiline comments are allowed, i.e. a multiline comment may, anywhere, contain a nested multiline comment, and the closing ’;;;}’ for the nested comment does not close the outer comment.

3.2. High-Level Syntax

3.2.1. Morpho Program



Syntax Diagram 44: ⟨*program*⟩

Syntax diagram 44 shows the syntax of a Morpho program, i.e. a compilation unit. A Morpho compilation unit is a sequence of compilation commands. A compilation command does one of the following:

- Compiles a module into a file.
- Compiles a module into a module variable.
- Compiles a program into a file.
- Shows the contents of a module.

For example, the following code creates the module file `reverse.mmod` which contains the single function `reverse`. The last line of this code is a show command that causes a description of the module to be written to the screen during compilation.

```
"reverse.mmod" =
{{
reverse =
        fun(x)
        {
                var y=[];
                while( x )
                {
                        y = head(x) : y;
                        x = tail(x);
                };
                y;
        };
}};

show "reverse.mmod";
```

Similarly, the beginning of the following code creates the module variable `rev` which contains the same module as above, which can then be used subsequently in the same code file, but will be forgotten once the compilation of that file finishes. The module variable is then used, in this example, in the creation of an executable file, `rev.mexe`.

```
rev =
{{
reverse =
        fun(x)
        {
                var y=[];
                while( x )
                {
                        y = head(x) : y;
                        x = tail(x);
                };
                y;
        };
}};

"rev.mexe" = main in
{{
main =
        fun()
        {
                writeln(reverse([1,2,3,4]));
        };
}}
*
```

```
25 rev
26 *
27 BASIS
28 ;
```

The same executable could have been created as follows, assuming the existence of the module file `reverse.mmod`.

```
1  "rev.mexe" = main in
2  {{
3  main =
4          fun()
5          {
6                  writeln(reverse([1,2,3,4]));
7          };
8  }}
9  *
10 "reverse.mmod"
11 *
12 BASIS
13 ;
```

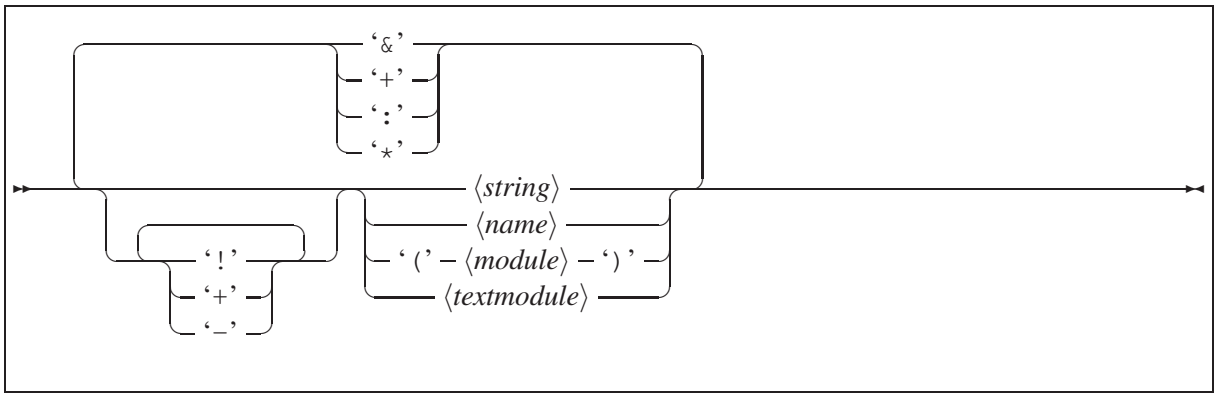
3.2.2. Modules

Syntax diagram 45 on the next page shows the syntax of a Morpho module. As the diagram indicates, modules can be composed using the binary operations '&', '+', ':', and '*', and they can be operated on by the unary operations '!', '+', and '-'. The unary operations have the highest precedence. Among the binary operations the '&' operation (link operation) has the lowest precedence. The '+' operation (join operation) has the next higher, the ':' operation (compose operation) the next, and finally the '*' (importation operation) has the highest precedence among the binary operations. Parentheses can be used to modify the application of the operations in the usual fashion.

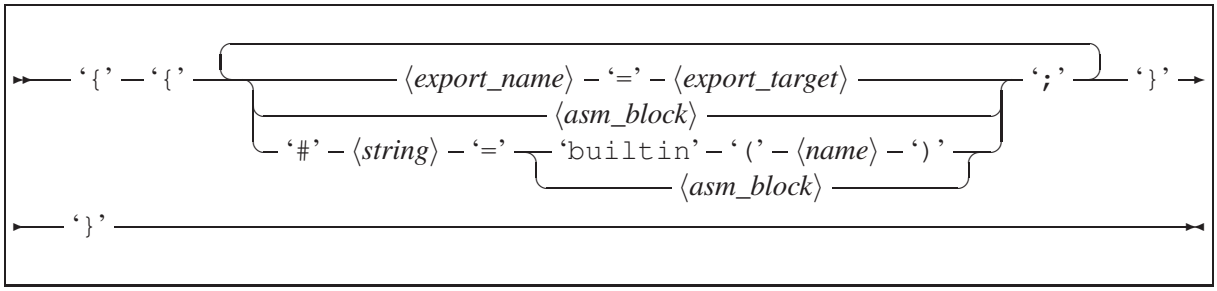
The semantics of the module operations are described elsewhere.

The simplest modules are:

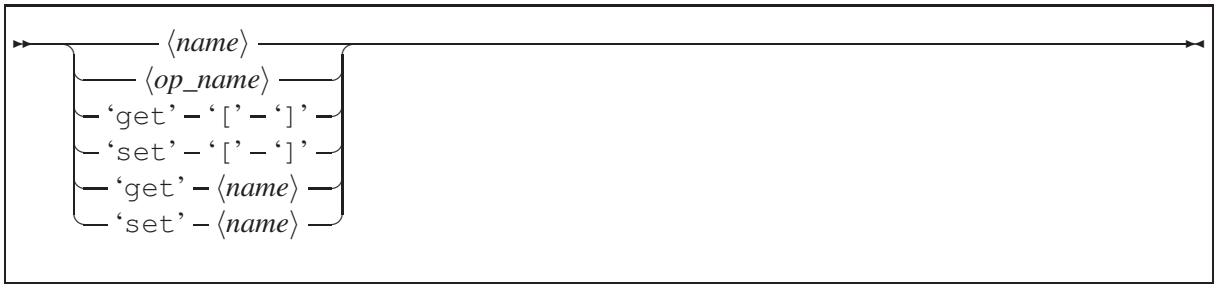
- A string, which represents a preexisting modules loaded from a file.
- A name, which represents a preexisting module loaded from a module variable.
- A text module, which represents a module created from scratch with Morpho source code or with Morpho assembly language, the semantics of which is not described in this manual.



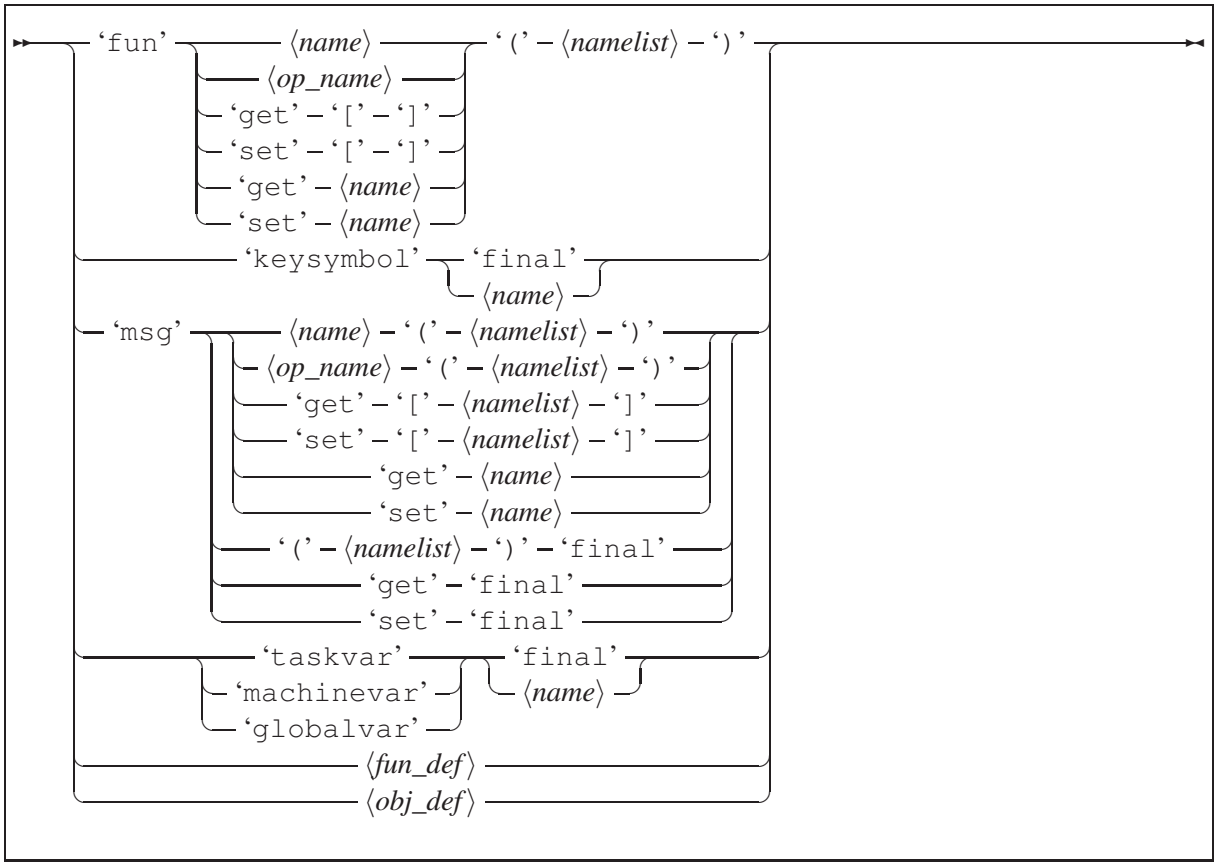
Syntax Diagram 45: ⟨*module*⟩



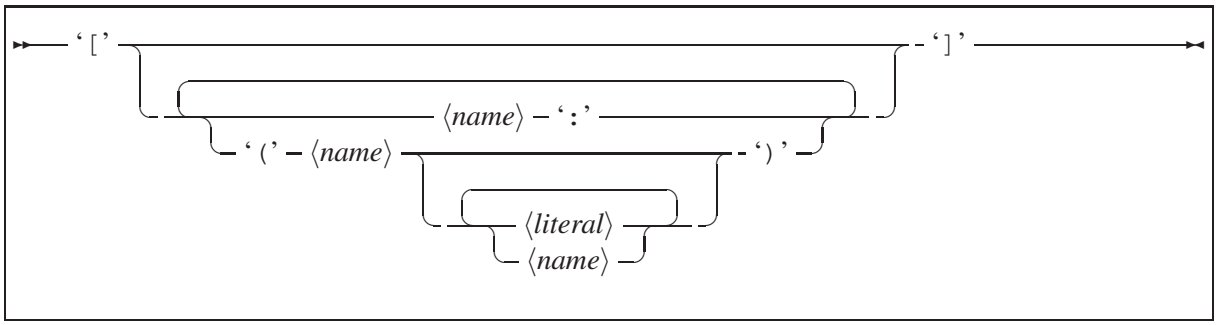
Syntax Diagram 46: ⟨*textmodule*⟩



Syntax Diagram 47: ⟨*export_name*⟩



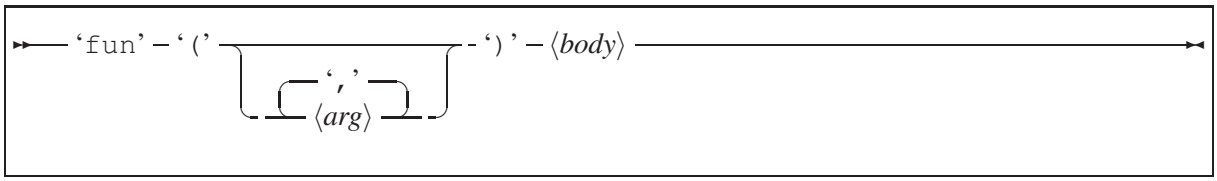
Syntax Diagram 48: ⟨*export_target*⟩



Syntax Diagram 49: ⟨*asm_block*⟩

3.2.3. Function Definitions

Syntax diagram 50 shows the syntax of a function definition. This syntax pattern describes both top-level functions that are exported from a module and nested functions inside other functions or object definitions. As with functions in most other programming languages Morpho functions are invoked using call expressions (described elsewhere) which cause the function to be executed.

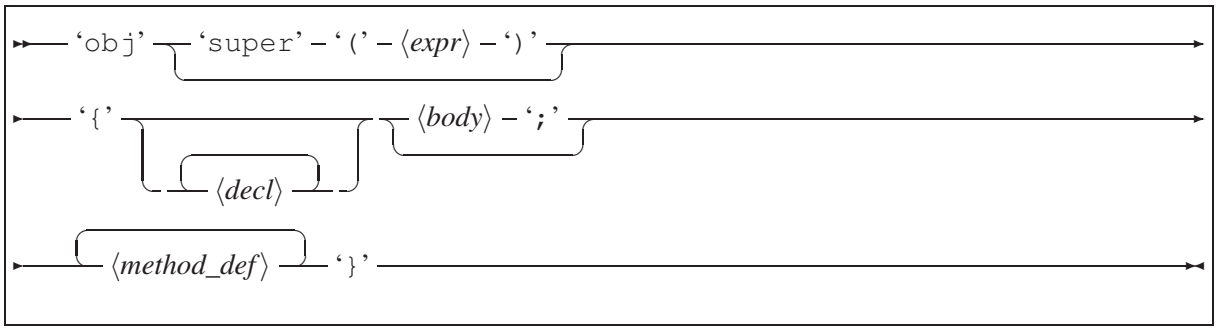


Syntax Diagram 50: ⟨*fun_def*⟩

3.2.4. Object Definitions and Constructor Definitions

Syntax diagram 51 shows the syntax of a Morpho object definition. This is an expression that returns a new Morpho object.

Syntax diagram 52 on page 68 shows the syntax of a Morpho object constructor definition. This syntax pattern describes both top-level object definitions that are exported from a module and also nested object definitions inside other functions or object definitions. A Morpho object constructor is invoked as a function call, which causes a new instance of a corresponding object to be created and returned.



Syntax Diagram 51: ⟨*object_def*⟩

The following contrived *hello world* example code creates an object and sends a message to it.

```
1 val x =
2   obj
3   {
4   msg hello()
5     {
6       writeln("Hello World")
7     }
```

```
  };
x.hello();
```

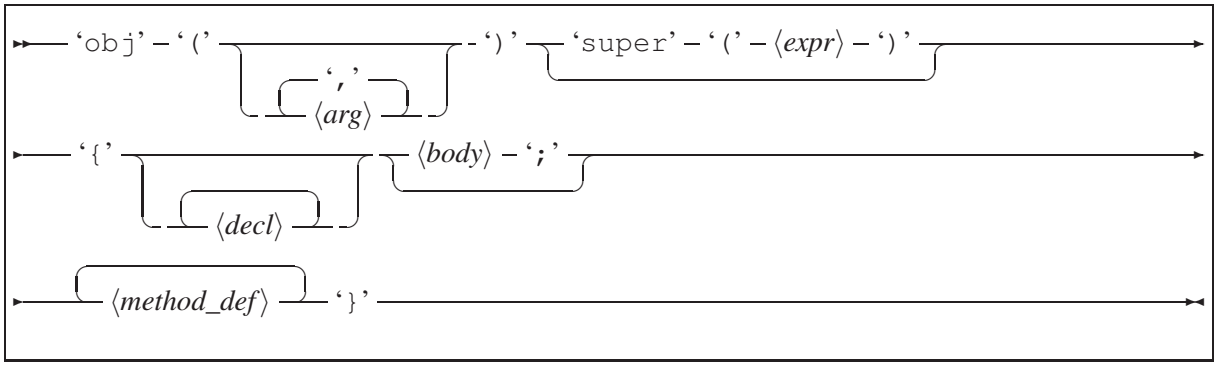
The following code has the same effect, but uses a constructor with an argument.

```
rec
  obj xobj(str_in)
  {
  val str = str_in;
  msg hello()
    {
      writeln(str)
    }
  };
val x = xobj("Hello World");
x.hello();
```

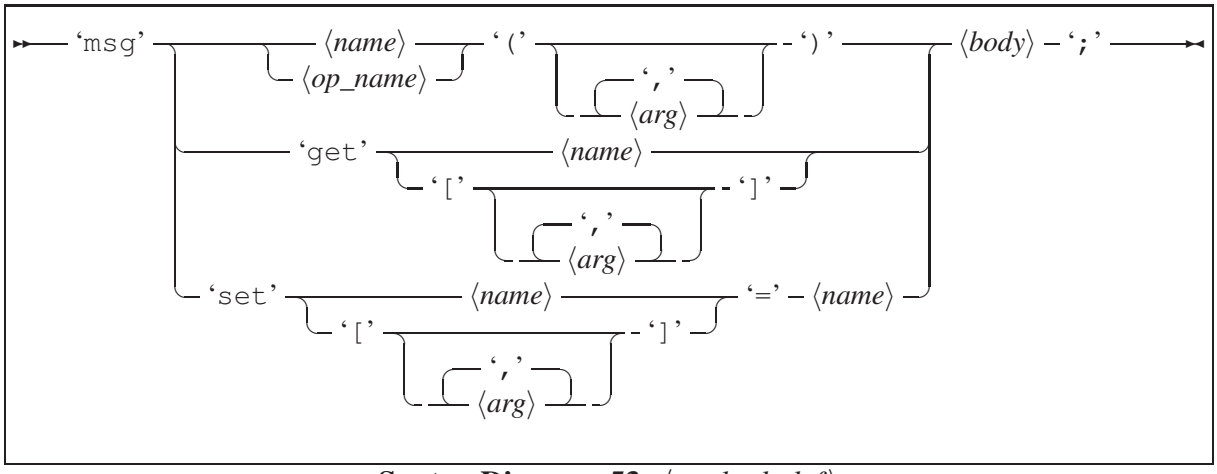
Note that the argument, `str\_in`, to the constructor is assigned to an instance variable, `str`. Constructor arguments are only visible during construction.

The following example shows how to do the same thing in compilable Morpho code.

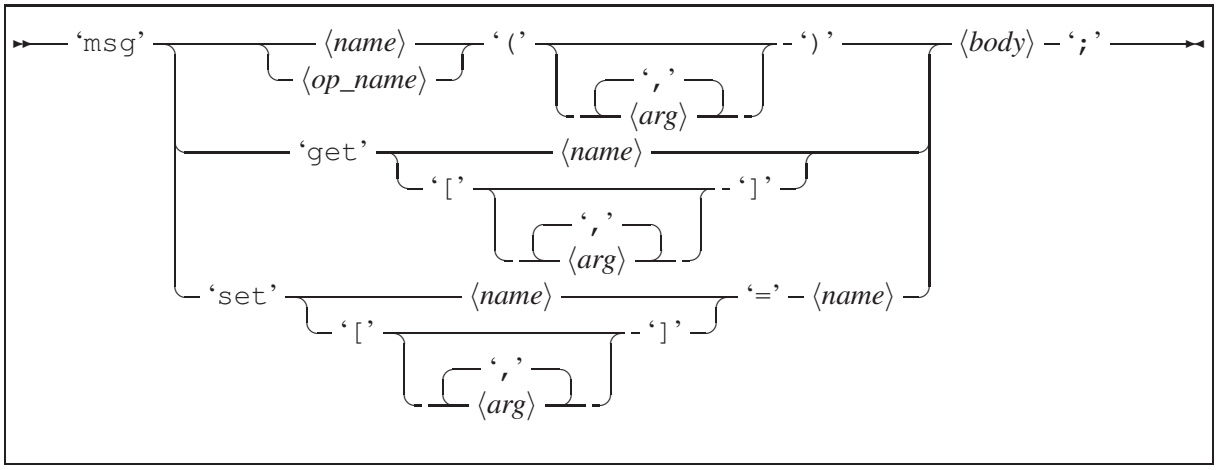
```
"test.mexe" = main in
!
{{
main =
  {
    val x = xobj("Hello World");
    x.hello();
  };

xobj =
  obj(str_in)
  {
  val str = str_in;
  msg hello()
    {
      writeln(str)
    }
  };
}}
*
BASIS
;
```



Syntax Diagram 52: ⟨*constructor_def*⟩



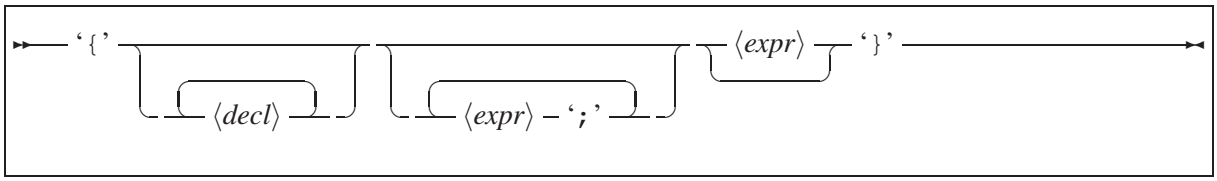
Syntax Diagram 53: ⟨*method_def*⟩



Syntax Diagram 54: ⟨*method_def*⟩

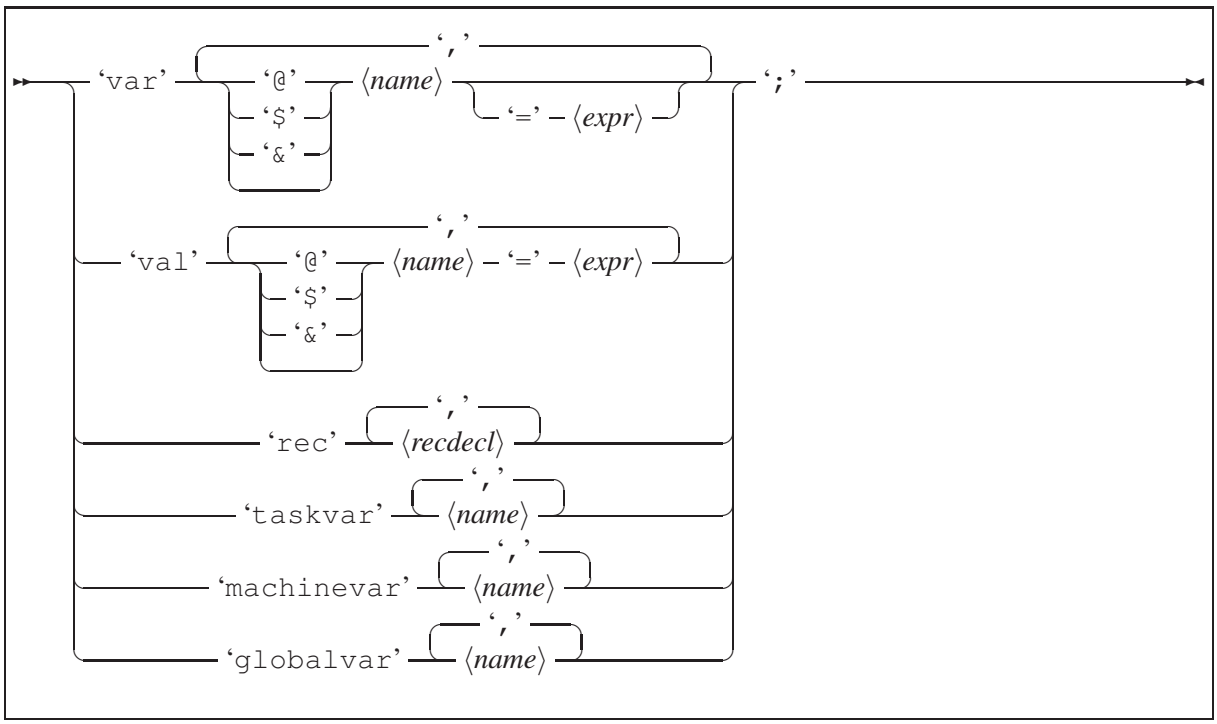
3.2.5. Body

Syntax diagram 55 shows the syntax of a Morpho code body. Such bodies are parts of various syntax patterns and stand for an executable sequence of expressions, optionally preceeded by declarations.



Syntax Diagram 55: ⟨*body*⟩

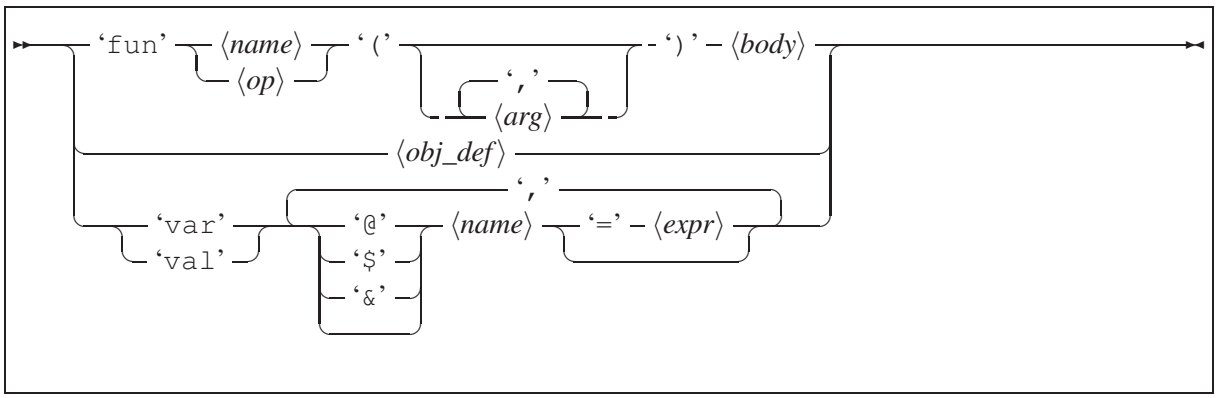
3.2.6. Declarations



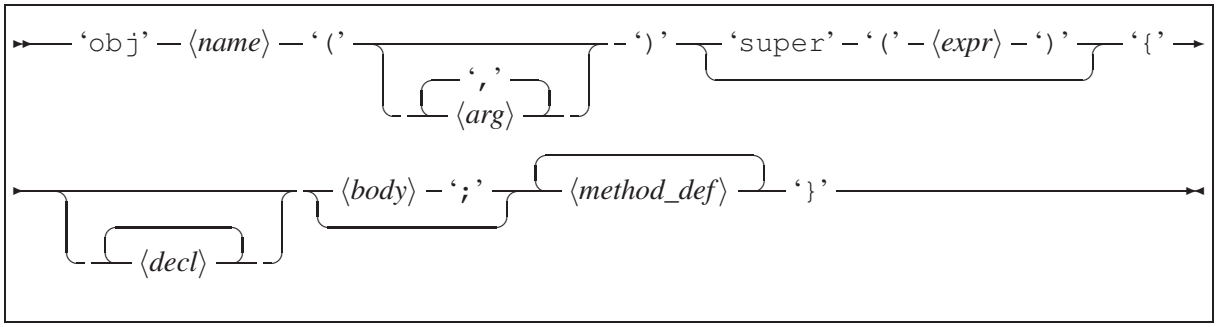
Syntax Diagram 56: ⟨*decl*⟩

3.2.7. Expressions

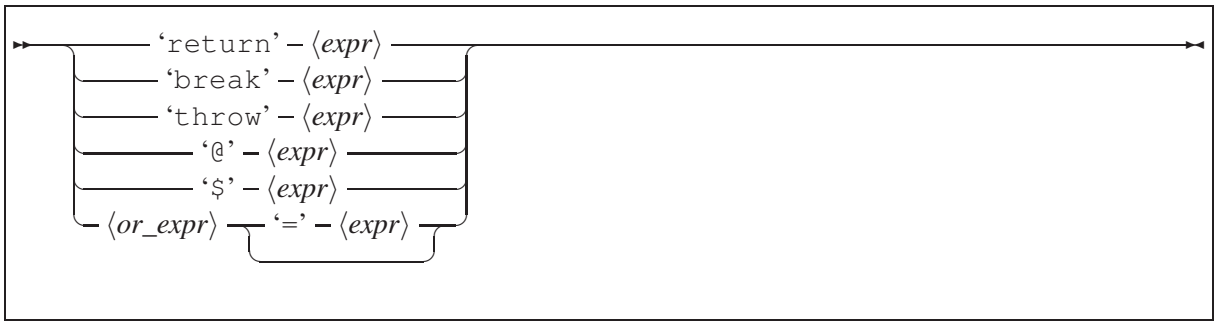
Note that the only semantically valid assignment targets are variables (⟨*name*⟩), fields (⟨*expr*⟩.⟨*name*⟩ and ⟨*expr*⟩.#⟨*name*⟩ and ⟨*expr*⟩.#⟨*string*⟩ and ⟨*expr*⟩.##⟨*name*⟩ and ⟨*expr*⟩.##⟨*string*⟩), and array items (⟨*expr*⟩[...]). The compiler will generate an error if the target of an assignment is not semantically valid.



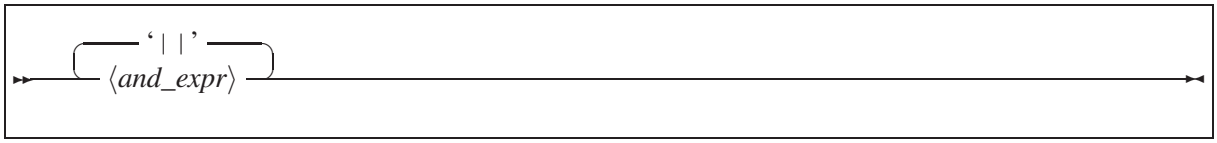
Syntax Diagram 57: ⟨*recdecl*⟩



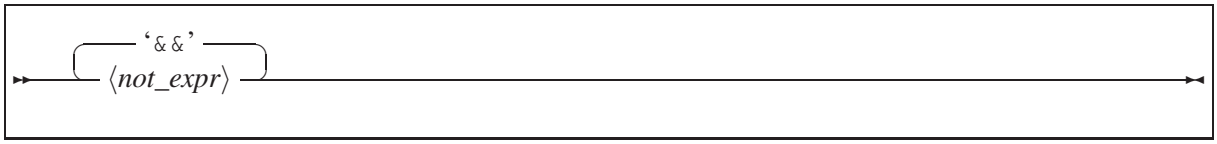
Syntax Diagram 58: ⟨*obj_def*⟩



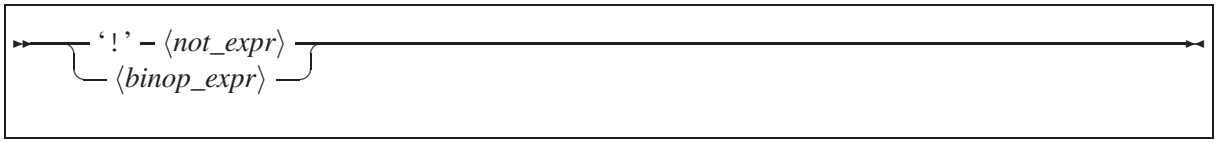
Syntax Diagram 59: ⟨*expr*⟩



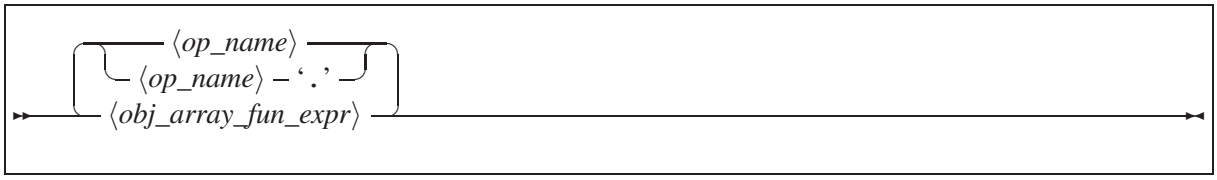
Syntax Diagram 60: ⟨*or_expr*⟩



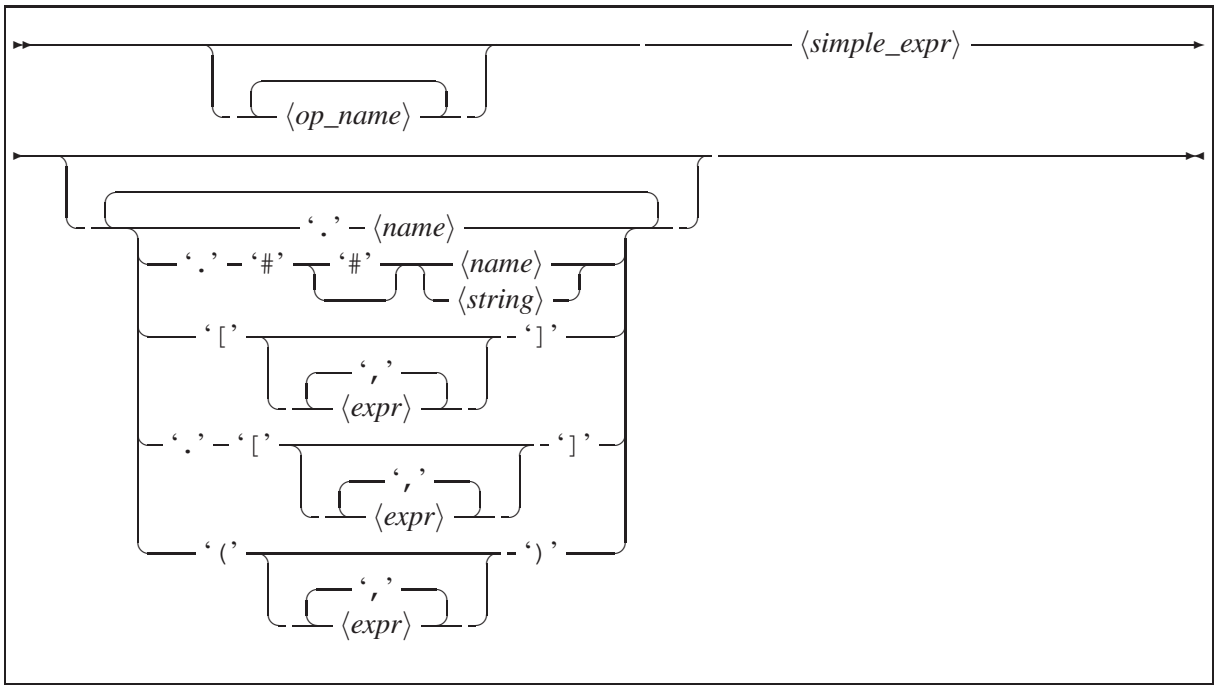
Syntax Diagram 61: ⟨*and_expr*⟩



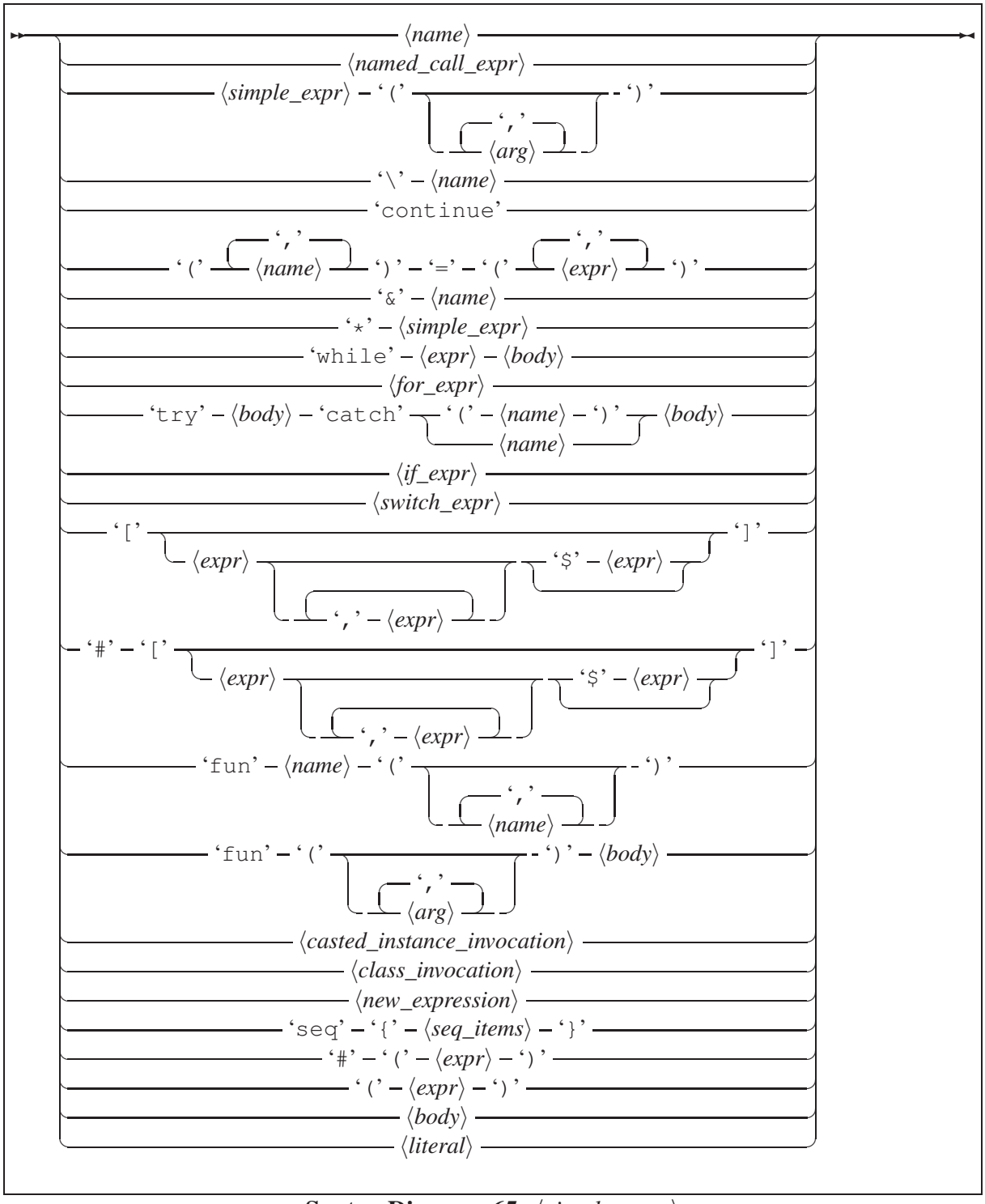
Syntax Diagram 62: ⟨*not_expr*⟩



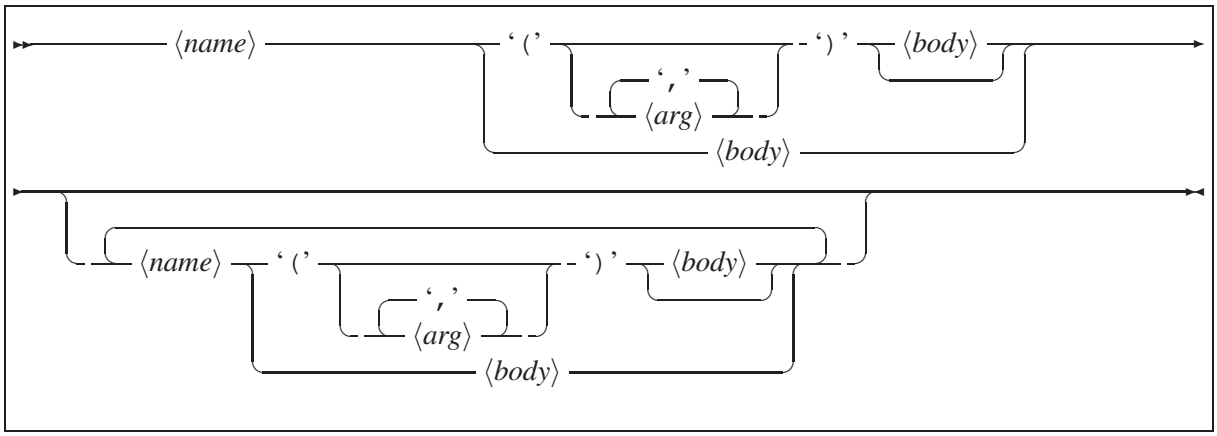
Syntax Diagram 63: ⟨*binop_expr*⟩



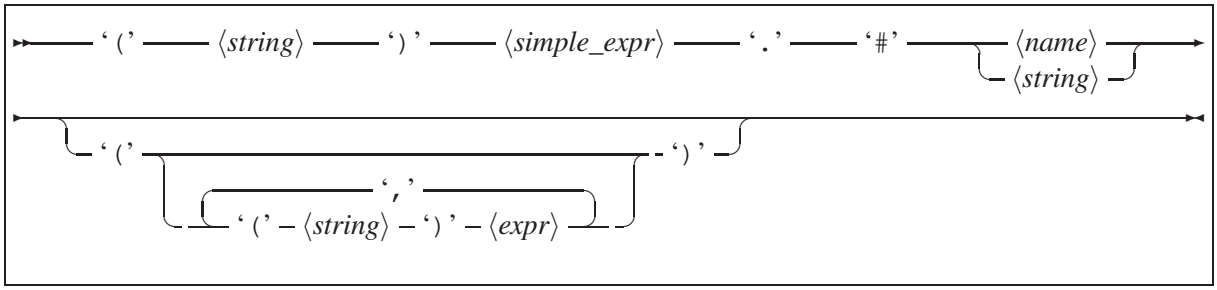
Syntax Diagram 64: ⟨*obj_array_fun_expr*⟩



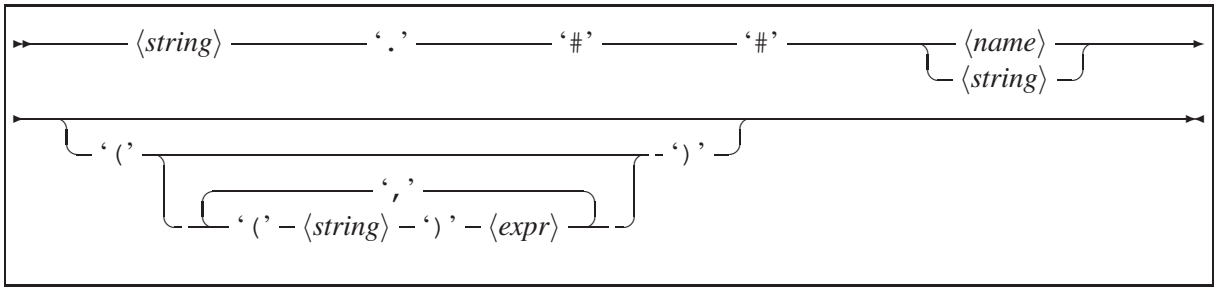
Syntax Diagram 65: ⟨*simple_expr*⟩



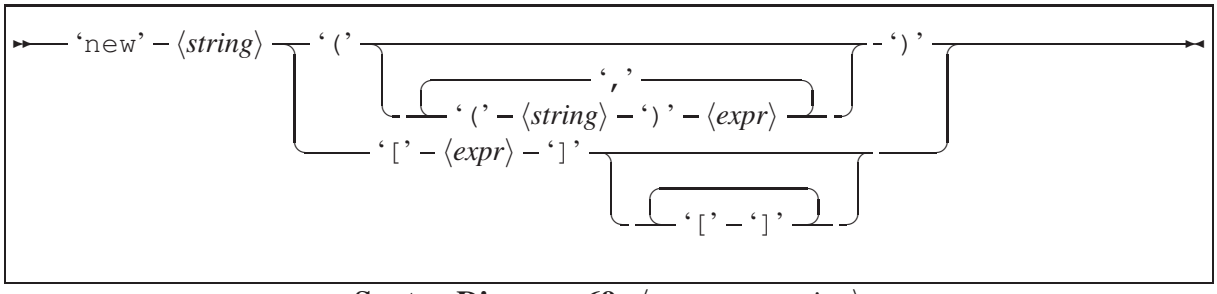
Syntax Diagram 66: ⟨*named_call_expr*⟩



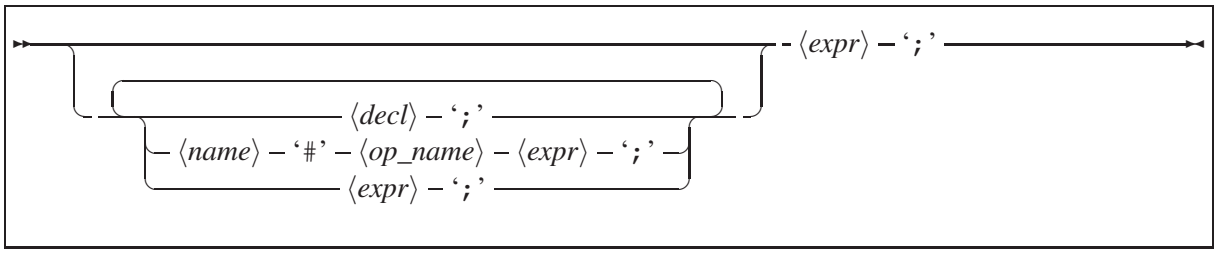
Syntax Diagram 67: ⟨*casted_instance_invocation*⟩



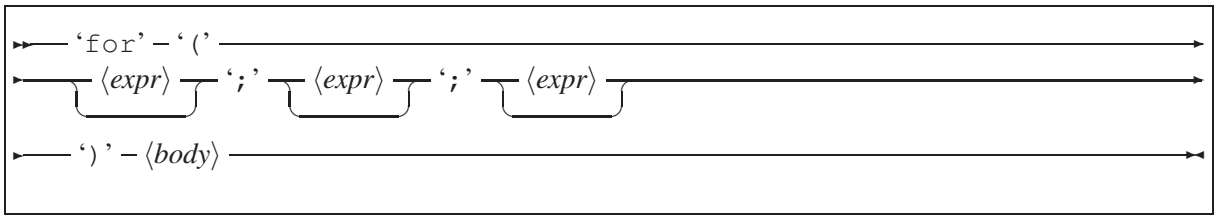
Syntax Diagram 68: ⟨*class_invocation*⟩



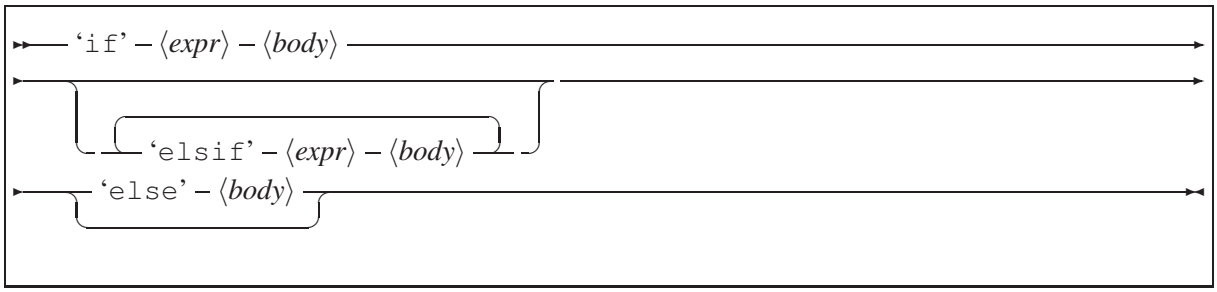
Syntax Diagram 69: ⟨*new_expression*⟩



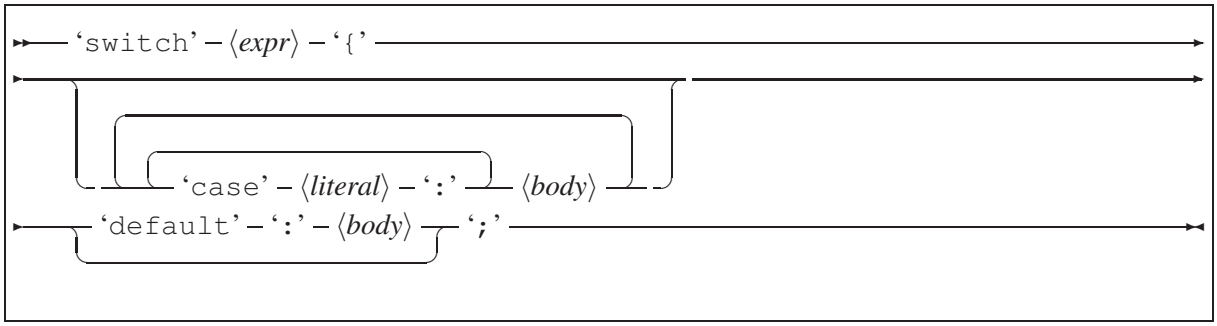
Syntax Diagram 70: ⟨*seq_items*⟩



Syntax Diagram 71: ⟨*for_expr*⟩



Syntax Diagram 72: ⟨*if_expr*⟩



Syntax Diagram 73: ⟨*switch_expr*⟩

4. Modules

Morpho has a unique way of handling modules. In Morpho we consider a module to be a first order substitution mapping names to values. Each module maps a finite set of names. A name can be mapped to one of the following:

- a function,
- an object constructor (actually an object constructor is a function),
- a message,
- a static variable, or
- another name.

4.1. Importation

Figure 4.1 on the following page shows a general example of module importation.

4.2. Join

Figure 4.2 on the next page shows a general example of module join.

4.3. Iteration

Figure 4.3 on page 77 shows a general example of module iteration.

4.4. Composition

Figure 4.4 on page 77 shows a general example of module composition.

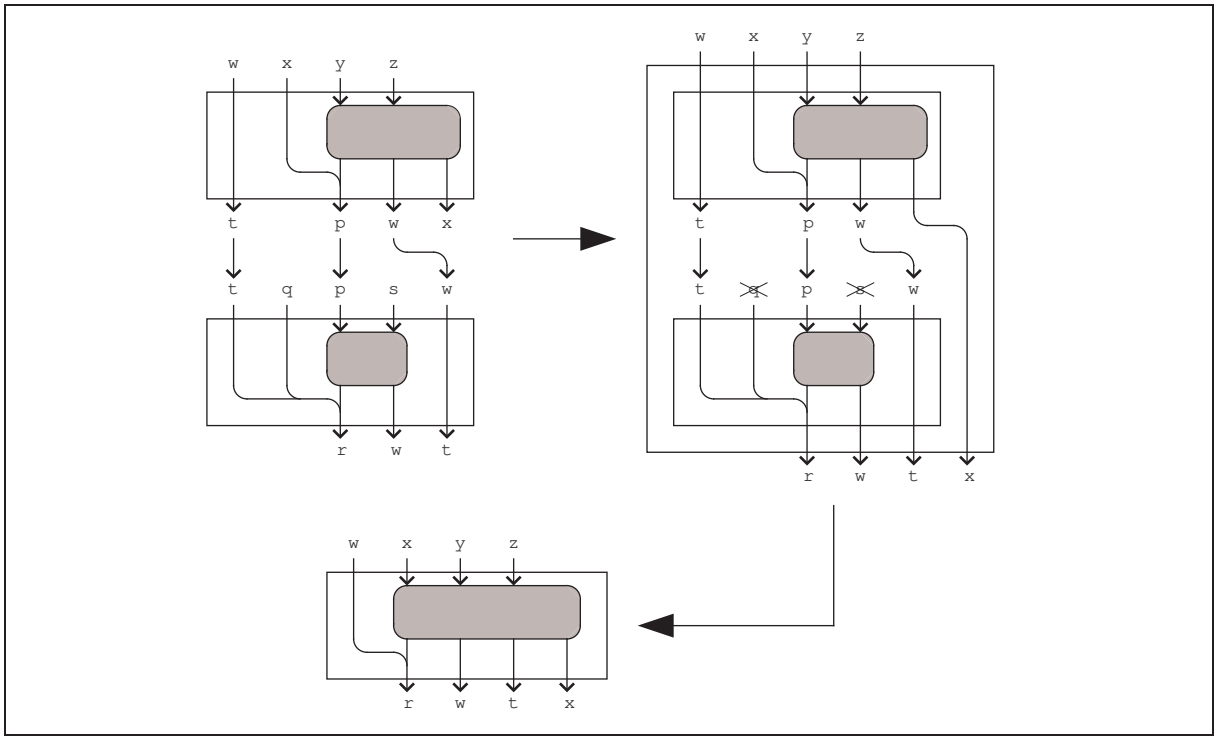


Figure 4.1.: Module Importation

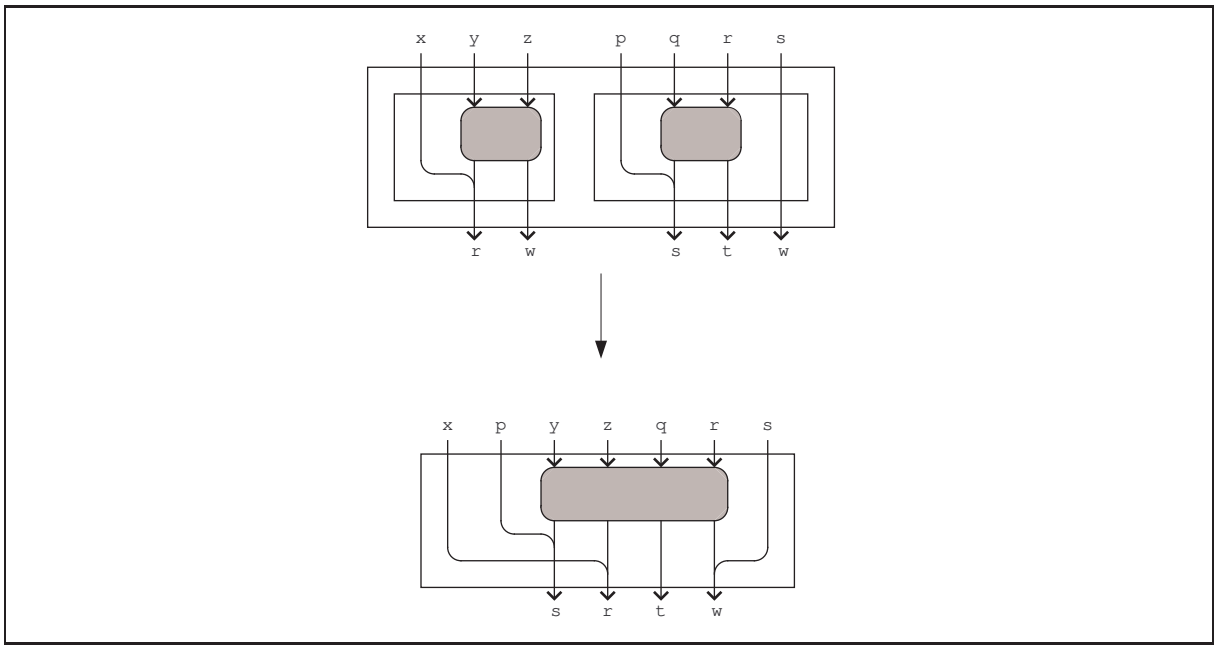


Figure 4.2.: Module Join

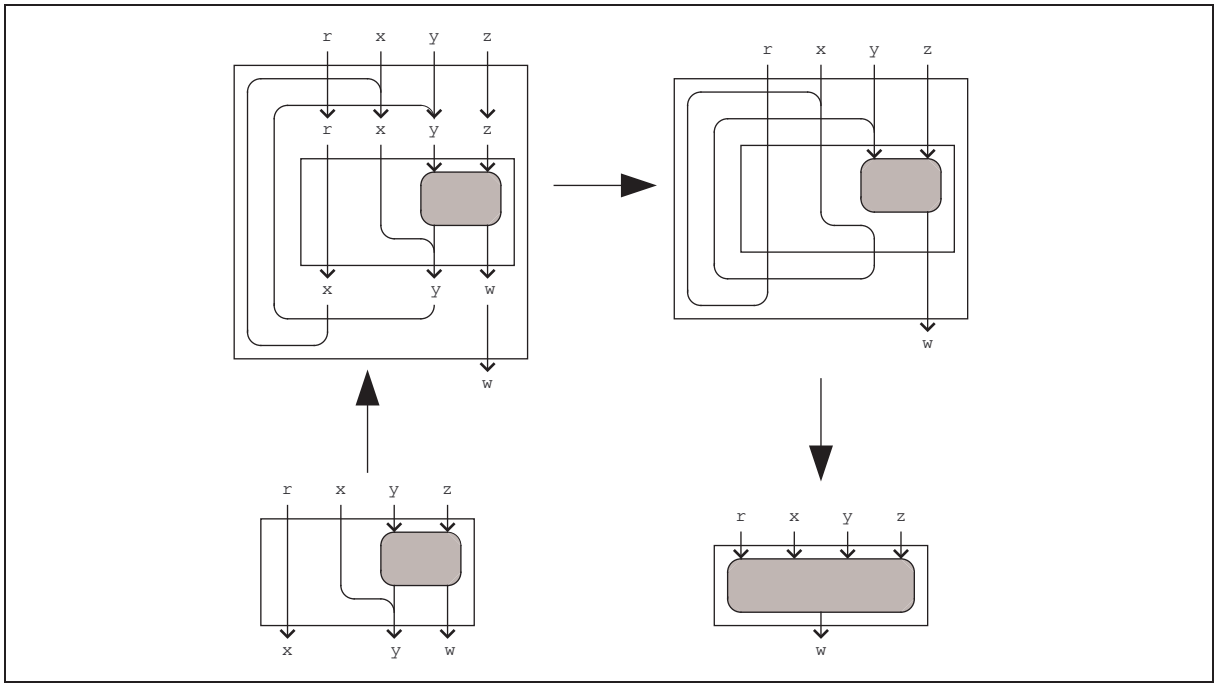


Figure 4.3.: Module Iteration

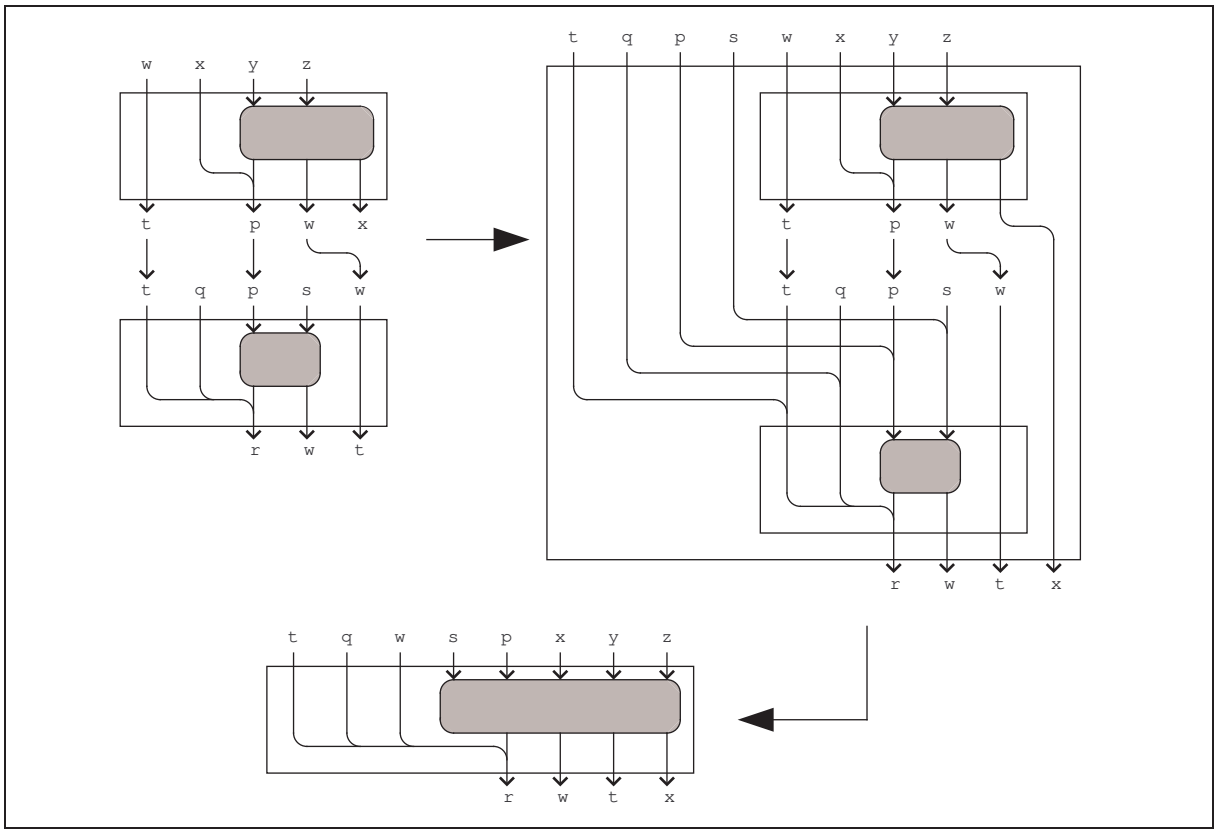


Figure 4.4.: Module Composition

Part II.

The BASIS Module

-

Use: `z = x - y;`

Pre: `x` and `y` should be numbers.

Post: `z` is the result of subtracting `y` from `x`. If both are integers, the result is an integer. If both are double, the result is double.

-

Use: `y = - x;`

Pre: `x` should be a number.

Post: `y` is the result of subtracting `x` from zero. If `x` is an integer, the result is an integer. If `x` is a double, the result is double.

!=

Use: `b = x != y;`

Pre: `x` and `y` can be any values.

Post: `b` is true if one is null but not the other or if they are objects that are not equal.

%

Use: `z = x % y;`

Pre: `x` and `y` should both be integers and `y` should be positive.

Post: `z` is the remainder when `x` is divided by `y`.

*****

Use: `z = x * y;`

Pre: `x` and `y` should be numbers.

Post: `z` is the result of multiplying `y` and `x`. If both are integers, the result is an integer. If both are double, the result is double.

/

Use: `z = x / y;`

Pre: `x` and `y` should be numbers, `y` should not be zero.

Post: `z` is the result of dividing `x` by `y`. If both are integers, the result is an integer. If both are double, the result is double.

:

Use: `z = x : y;`

Pre: `x` and `y` can be any values.

Post: `z` is a new pair with `x` as head and `y` as tail.

??

Use: `z = x ?? y;`

Pre: `x` and `y` can be any values.

Post: `z` is the same as `x` unless `x` is **`null`**, in which case `z` is the same as `y`.

+

Use: `z = x + y;`

Pre: `x` and `y` should be numbers.

Post: `z` is the result of adding `y` and `x`. If both are integers, the result is an integer. If both are double, the result is double.

++

Use: `z = x ++ y;`

Pre: `x` should be a string and `y` can be any value.

Post: `z` is the result of appending `y` to `x`. If `y` is not a string it is converted to a string using the `toString()` Java method.

<-

Use: `c <- x;`

Pre: `c` is an open channel and `x` can be any value.

Post: The value `x` has been sent through the channel `c`.

<-

Use: `x = <- c;`

Pre: `c` is an open channel.

Post: The value `x` has been received through the channel `c`.

<

Use: `b = x < y;`

Pre: `x` and `y` should be Comparable and comparable to each other.

Post: `b` is true if `x` is less than `y`, false otherwise.

<=

Use: `b = x <= y;`

Pre: `x` and `y` should be Comparable and comparable to each other.

Post: `b` is true if `x` is less than or equal to `y`, false otherwise.

==

Use: `b = x == y;`

Pre: `x` and `y` can be any values.

Post: `b` is true if both are null or of they refer to equal objects, false otherwise.

>

Use: `b = x > y;`

Pre: `x` and `y` should be Comparable and comparable to each other.

Post: `b` is true if `x` is greater than `y`, false otherwise.

>=

Use: `b = x >= y;`

Pre: `x` and `y` should be Comparable and comparable to each other.

Post: `b` is true if `x` is greater than or equal to `y`, false otherwise.

>>=

Use: `y = x >>= f;`

Pre: `x` should be a stream of values, f should be a function of one argument that returns a stream.

Post: `y` is the stream of values that is composed by concatenating the streams that result from feeding the values in the stream `x` to the function `f`. See also the function `streamConcatMap`.

abs

Use: `y = abs(x);`

Pre: `x` should be a Double.

Post: `y` is the absolute value of `x`.

acos

Use: `y = acos(x);`

Pre: `x` should be a Double between -1.0 and 1.0, inclusive.

Post: `y` is the arc cosine of `x`, an angle between 0.0 and *pi*.

acquireMutex

Use: `acquireMutex(m);`

Pre: `m` is a mutex.

Post: The mutex has been acquired. If some other sequence of execution previously had possesion of the mutex then the current fiber waited until the mutex was released.

append

Use: `z = append(x,y);`

Pre: `x` and `y` are lists of any finite length.

Post: `z` contains a list starting with all the values from `x` which are then followed by `y`. Note that the pairs in `y` are shared between `z` and `y`, so, for example, `append([],y)` is exactly the same list as `y`, whereas `append(x,[])` has no top-level pair in common with `x` even though it contains exactly the same values in the same sequence.

Storing in Arrays

Use: `a[i] = x;`

Pre: `a` is either a Java Object array (Object[]) or a hashmap created by calling makeHashMap. If `a` is an Object array then `i` must be an integer value that is a valid index in that array. If `a` is a hashmap then `i` can be any value that fulfills the contracts of the Java hashCode and equals methods.

Post: The value of `x` has been stored in `a` under index `i`.

Fetching from Arrays

Use: `x = a[i];`

Pre: `a` is either an array (T[] for some type T) or a hashmap created by calling makeHashMap. If `a` is an Object array then `i` must be an integral value that is a valid index in that array. If `a` is a hashmap then `i` can be any value that fulfills the contracts of the Java hashCode() and equals() methods.

Post: The variable `x` contains the value stored in `a` under index `i`.

arrayGet

Use: `x = arrayGet(a,i);`

Pre: `a` is an array (T[] for some type T), i is an integer (byte, short, int, long or BigInteger).

Post: The variable `x` contains the value in position `i` of the array `a`.

arrayLength

Use: `n = arrayLength(a);`

Pre: `a` is an Object array (Object[]).

Post: The variable `n` contains the length of the array `a`.

arrayPut

Use: `arrayPut(a,i,x);`

Pre: `a` is an array (T[] for some type T), i is an integer (byte, short, int, long or BigInteger), `x` contains a value of type T.

Post: The the value in position `i` of the array `a` has been set to `x`.

Note: Same as arraySet.

arraySet

Use: `arraySet(a,i,x);`

Pre: `a` is an array (T[] for some type T), i is an integer (byte, short, int, long or BigInteger), `x` contains a value of type T.

Post: The the value in position `i` of the array `a` has been set to `x`.

Note: Same as arrayPut.

asin

Use: `y = asin(x);`

Pre: `x` should be a Double between -1.0 and 1.0, inclusive.

Post: `y` is the arc sine of `x`, an angle between -*pi*/2 and *pi*/2.

atan

Use: `y = atan(x);`

Pre: `x` should be a Double.

Post: `y` is the arc tangent of `x`, an angle between -*pi*/2 and *pi*/2.

atan

Use: `r = atan(y,x);`

Pre: `x` and `y` should be Double values, not both zero.

Post: `r` is the angle of the vector `(x,y)`, an angle between -*pi* and *pi*.

atan2

Use: `r = atan2(y,x);`

Pre: `x` and `y` should be Double values, not both zero.

Post: `r` is the angle of the vector `(x,y)`, an angle between -*pi* and *pi*.

Note: This is the same function as the binary `atan` function.

bigInteger

Use: `y = bigInteger(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same value, but as a BigInteger object.

byte

Use: `y = byte(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same value (if possible), but as a Byte object.

caaaar

Use: `y = caaaar(x);`

Pre: `x` must be a list of appropriate structure.

Post: `y` now contains the same value as would be returned by the call `car(car(car(car(x))))`.

caaadr
Similar to `caaaar` except equivalent to `car(car(car(cdr(x))))`.

caaar
Similar to `caaaar` except equivalent to `car(car(car(x)))`.

caadar
Similar to `caaaar` except equivalent to `car(car(cdr(car(x))))`.

caaddr
Similar to `caaaar` except equivalent to `car(car(cdr(cdr(x))))`.

caadr
Similar to `caaaar` except equivalent to `car(car(cdr(x)))`.

caar
Similar to `caaaar` except equivalent to `car(car(x))`.

cadaar
Similar to `caaaar` except equivalent to `car(cdr(car(car(x))))`.

cadadr
Similar to `caaaar` except equivalent to `car(cdr(car(cdr(x))))`.

cadar
Similar to `caaaar` except equivalent to `car(cdr(car(x)))`.

cadadr
Similar to `caaaar` except equivalent to `car(cdr(car(cdr(x))))`.

caddar
Similar to `caaaar` except equivalent to `car(cdr(cdr(car(x))))`.

cadddr
Similar to `caaaar` except equivalent to `car(cdr(cdr(cdr(x))))`.

caddr
Similar to `caaaar` except equivalent to `car(cdr(cdr(x)))`.

cadr
Similar to `caaaar` except equivalent to `car(cdr(x))`.

car
Same as the function `head`.

cdaaar
Similar to `caaaar` except equivalent to `cdr(car(car(car(x))))`.

cdaadr
Similar to `caaaar` except equivalent to `cdr(car(car(cdr(x))))`.

cdaar
Similar to `caaaar` except equivalent to `cdr(car(car(x)))`.

cdadar
Similar to `caaaar` except equivalent to `cdr(car(cdr(car(x))))`.

cdaddr
Similar to `caaaar` except equivalent to `cdr(car(cdr(cdr(x))))`.

cdadr
Similar to `caaaar` except equivalent to `cdr(car(cdr(x)))`.

cdar
Similar to `caaaar` except equivalent to `cdr(car(x))`.

cddaar
Similar to `caaaar` except equivalent to `cdr(cdr(car(car(x))))`.

cddadr
Similar to `caaaar` except equivalent to `cdr(cdr(car(cdr(x))))`.

cddar
Similar to `caaaar` except equivalent to `cdr(cdr(car(x)))`.

cdddar
Similar to `caaaar` except equivalent to `cdr(cdr(cdr(car(x))))`.

cddddr
Similar to `caaaar` except equivalent to `cdr(cdr(cdr(cdr(x))))`.

cdddr
Similar to `caaar` except equivalent to `cdr(cdr(cdr(x)))`.

cddr
Similar to `caaar` except equivalent to `cdr(cdr(x))`.

cdr
Same as the function `tail`.

canReadChannel

Use: `b = canReadChannel(c);`

Pre: `c` must be a channel (an is.hi.cs.morpho.Channel object).

Post: `b` now contains true iff the channel can be read from without blocking.

canWriteChannel

Use: `b = canWriteChannel(c);`

Pre: `c` must be a channel (an is.hi.cs.morpho.Channel object).

Post: `b` now contains true iff the channel can be written to without blocking.

cbrt

Use: `y = cbrt(x);`

Pre: `x` should be a Double.

Post: `y` is the cube root of `x`.

channelEOF

Use: `eof = channelEOF();`

Post: `eof` contains a special channelEOF value that represents enf-of-file on a channel.

char

Use: `y = char(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same ordinal value (if possible), but as a Character object.

closeChannel

Use: `closeChannel(c);`

Pre: `c` must be a channel.

Post: `c` is now closed. All current and subsequent writes to the channel will immediately succeed without effect and all current and subsequent reads will immediately return the channelEOF value. See also the BASIS function `channelEOF`.

cos

Use: `x = cos(r);`

Pre: `r` should be a Double.

Post: `x` is the cosine of `r`.

dec

Use: `y = dec(x);`

Pre: `x` is an integer value.

Post: The variable `y` contains `x-1`.

double

Use: `y = double(x);`

Pre: `x` must be a Float, Double, BigInteger, Long, Integer, Short, Character or a Byte.

Post: `y` now contains the same value (as far as possible), but as a Double object.

eq

Use: z = eq(x,y);|

Pre: `x` and `y` are any values.

Post: `z` is true if and only if either both `x` and `y` are **`null`** or if `x` and `y` refer to the same object. If `x` and `y` refer to different objects then `z` is false, even if the objects are equal.

exit

Use: exit(n);|

Pre: `n` is an integer value.

Post: The program has exited with exit code `n`.

exp

Use: `y = exp(x);`

Pre: `x` should be a Double.

Post: `y` is *e* raised to the power `x`, e^x.

expm1

Use: `y = expm1(x);`

Pre: `x` should be a Double.

Post: `y` is *e* raised to the power `x`, minus 1, $e^x - 1$.

filterList

Use: `y = filterList(p,x);`

Pre: `x` is a list, `p` is a unary predicate that maps each value in `x` to a truth value.

Post: `y` contains a list of the values `z` from `x` such that `p(z)` is true.

Example: `filterList(fun(z){z\%2==0},[1,2,3,4])` returns the list `[2,4]`.

filterStream

Use: `y = filterStream(p,x);`

Pre: `x` is a stream (may be infinite), `p` is a unary predicate that maps each value in `x` to a truth value.

Post: `y` contains a stream of the values `z` from `x` such that `p(z)` is true.

forAll_inList_do

Use: `forAll_inList_do(@x,y,@b);`

Pre: `x` is a variable (by name), `y` is a list, `b` is an executable body (by name).

Post: The body `b` has been executed once for each value in the list `y`. Before each execution the variable `x` is given the value from the list.

Example: The code

```
1 forAll( var x )
2 inList( [1,2,3] )
3 do
4 {
5   writeln(x)
6 }
```

writes three lines with the values 1, 2, and 3. The code

```
1 {
2   var x;
3   forAll_inList_do(@x,[1,2,3],@{writeln(x)})
4 }
```

does exactly the same.

forAll_inStream_do

Use: `forAll_inStream_do(@x,y,@b);`

Pre: `x` is a variable (by name), `y` is a stream, `b` is an executable body (by name).

Post: The body `b` has been executed once for each value in the stream `y`. Before each execution the variable `x` is given the value from the stream.

Example: The code

```
1 forAll( var x )
2 inStream( #[1,2,3] )
3 do
4 {
5   writeln(x)
6 }
```

writes three lines with the values 1, 2, and 3. The code

```
1 {
2   var x;
3   forAll_inStream_do(@x,#[1,2,3],@{writeln(x)})
4 }
```

does exactly the same.

forAll_inChannel_do

Use: `forAll_inChannel_do(@x,y,@b);`

Pre: `x` is a variable (by name), `y` is a channel, `b` is an executable body (by name).

Post: The body `b` has been executed once for each value in the channel `y`. Before each execution the variable `x` is given the value from the channel.

forAllKeys_inMap_do

Use: `forAllKeys_inMap_do(@x,y,@b);`

Pre: `x` is a variable (by name), `y` is a map[1] (for example a result from `makeHashMap()`), `b` is an executable body (by name).

Post: The body `b` has been executed once for each key in the map `y`. Before each execution the variable `x` is given the value of the key.

flushChannel

Use: `flushChannel(c);`

Pre: `c` is a writable channel.

Post: The channel has been flushed so that all the written values have been made readable by whatever fibers are trying to read from the channel.

parseInt

Use: `i = parseInt(s);`

Pre: `s` is a string that contains the text representation of an integer.

Post: `i` contains the integer as an Integer object.

parseLong

Use: `i = parseLong(s);`

Pre: `s` is a string that contains the text representation of an integer.

Post: `i` contains the integer as a Long object.

[1] A map is any object of a Java class that implements the java.util.Map interface.

go
Same as `goTask`.

goFiber

Use: `goFiber(@b);`
or (equivalent)
`goFiber{b};`

Pre: `b` is an executable body (by name).

Post: The body `b` has been started as a new fiber in the current task.

goMachine

Use: `goMachine(@b);`
or (equivalent)
`goMachine{b};`

Pre: `b` is an executable body (by name).

Post: The body `b` has been started as a new fiber in a new task in a new machine.

goTask

Use: `goTask(@b);`
or (equivalent)
`goTask{b};`

Pre: `b` is an executable body (by name).

Post: The body `b` has been started as a new fiber in a new task in the current machine.

force

Use: `x = force(y);`

Pre: `y` is a promise.

Post: `x` is the value of the promise. If the promise had not been forced before then the underlying expression has now just been evaluated once to get the value. If the promise was evaluated earlier then the value returned by that evaluation is returned again. Note that if multiple fibers or tasks force the same promise, only one of them will evaluate the underlying expression.

Example:

```
1   x = force($(i+1));
2   ;;; equivalent to: x = i+1;
```

forkFiber

Use: `s = forkFiber();`

Pre: Nothing.

Post: The current fiber has been forked into two fibers, both of which run in the same original task. One fiber continues running, the other one becomes one of the ready fibers in the current task and will eventually run if the other fibers yield control and if the task is not killed before. In the fiber that continues running `s` is **`null`**, in the other fiber `s` refers to the current task (an object of type is.hi.cs.morpho.Task).

forkMachine

Use: `s = forkMachine();`

Pre: Nothing.

Post: The current fiber has been forked into two fibers, one of which runs in a new task in a new machine, the other one uninterrupted in the old task. In the fiber that continues running in the old task, `s` is **`null`**, in the other fiber `s` refers to the new task (an object of type is.hi.cs.morpho.Task).

forkSuspendedFiber

Use: `s = forkSuspendedFiber(&f,x);`

Pre: `&f` is a reference to a variable, `x` is any value.

Post: The current fiber has been forked into two fibers. More precisely one of them is really only a continuation instead of a full fiber. One of the fibers continues running, uninterrupted. In that fiber the return value, `s`, is the same value as `x`. The other fiber has been saved as a continuation in the variable `f` and can be activated as a fiber by calling the function `transfer`.

Note: `forkSuspendedFiber(&f,x)` is functionally equivalent to `transfer(&f,null,x)`.

Example: The following code will fork a suspended fiber (a continuation) and then transfer control to it. The effect is to write the string `"Hello world"` using a very roundabout method.

```
1         var f;
2         if( forkSuspendedFiber(&f,false) )
3         {
4                 writeln("Hello world");
5         }
6         else
```

```
7           {
8                   transfer(null,f,true);
9           };
```

forkTask

Use: `s = forkTask();`

Pre: Nothing.

Post: The current fiber has been forked into two fibers, one of which runs in a new task, the other one uninterrupted in the old task. In the fiber that continues running in the old task, `s` is **`null`**, in the other fiber `s` refers to the new task (an object of type is.hi.cs.morpho.Task).

format

Use: `s = format(f,v);`

Pre: `f` is a format string (see java.util.Formatter) and `v` is a value that the format string can format.

Post: `s` is a string that is the result of using the format string to format the value.

format

Use: `s = format(f,v1,v2);`

Pre: `f` is a format string (see java.util.Formatter), and `v1` and `v1` are values that the format string can format.

Post: `s` is a string that is the result of using the format string to format the values.

format

Use: `s = format(f,v1,v2,v3);`

Pre: `f` is a format string (see java.util.Formatter), and `v1`, `v2` and `v3` are values that the format string can format.

Post: `s` is a string that is the result of using the format string to format the values.

formatArray

Use: `s = format(f,arr);`

Pre: `f` is a format string (see java.util.Formatter), and `arr` is an Object array containing values that the format string can format.

Post: `s` is a string that is the result of using the format string to format the values.

from

Use: `s = from(i);`

Pre: `i` is an integer value.

Post: `s` is an infinite stream of the integer values from `i`. More precisely, `s` is the ascending stream of the integers `i`, `i+1`, ...

fromBy

Use: `s = fromBy(i,k);`

Pre: `i` and `k` are integer values.

Post: `s` is the infinite stream of the integers `i`, `i+k`, `i+k+k`, ...

fromByUpTo

Use: `s = fromByUpTo(i,k,j);`

Pre: `i`, `k` and `j` are integer values.

Post: `s` is a stream of the integer values less than or equal to `j` from the sequence `i`, `i+k`, `i+k+k`, ...

Example:

```
1   ;;; Write 1,3,5,7,9
2   var s = fromByUpTo(1,2,10);
3   while( s!=#[] )
4   {
5      writeln(streamHead(s));
6      s = streamTail(s);
7   };
```

fromUpTo

Use: `s = fromUpTo(i,j);`

Pre: `i` and `j` are integer values.

Post: `s` is a stream of the integer values from `i` to `j`, inclusive. More precisely, `s` is the ascending stream of integers k fulfilling $\texttt{i} \leq k \leq \texttt{j}$.

Example:

```
;;; Write 1..10
var s = fromUpTo(1,10);
while( s!=#[] )
{
   writeln(streamHead(s));
   s = streamTail(s);
};
```

getArgs

Use: `args = getArgs();`

Post: `args` is a String array containing the command-line arguments of the currently running program.

getExceptionTrace

Use: `t = getExceptionTrace(&e);`

Pre: `&e` is a reference to the receiving variable in a catch clause of a try expression.

Post: `t` contains the trace for the exception.

Example:

```
        try
        {
                ...
        }
        catch(e)
        {
                val t = getExceptionTrace(&e);
                ...
        };
```

getOutputChannel

Use: `c = getOutputChannel();`

Post: `c` contains the current output channel that receives output from calls to `write()` and `writeln()`.

getReturnContinuation

Use: `c = getReturnContinuation();`

Post: `c` contains the return continuation of the current function.

getTask

Use: `t = getTask();`

Post: `t` contains the current task.

getThunkGetter

Use: `s = getThunkGetter(t);`

Pre: `t` must be a thunk.

Post: `s` contains a closure that is the getter function of the thunk.

getThunkSetter

Use: `s = getThunkSetter(t);`

Pre: `t` must be a thunk.

Post: `s` contains a closure that is the setter function of the thunk.

head

Use: `y = head(x);`

Pre: `x` must be a pair.

Post: `y` contains the head of `y`.

Example: The code

```
1 {
2   val x=[1,2];
3   writeln(head(x));
4 }
```

will write the number 1.

hypot

Use: `z = hypot(x,y);`

Pre: `x` and `y` should be Double values.

Post: `z` is the length of the vector `(x,y)`.

inc

Use: `y = inc(x);`

Pre: `x` must be an integer value, i.e. a Byte, Short, Integer, Long or BigInteger.

Post: `y` contains the result from incrementing `x`, a value of the same type as `x`.

Example: The code

```
1 {
2   writeln(inc(0));
3 }
```

will write the number 1.

int

Use: `y = int(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same value (if possible), but as an Integer object.

isArray

Use: `b = isArray(a);`

Pre: `a` is any value.

Post: `b` now contains true iff `a` is an array.

isBigInteger

Use: `b = isBigInteger(i);`

Pre: `i` is any value.

Post: `b` now contains true iff `i` is a BigInteger.

isBoolean

Use: `b = isBoolean(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Boolean.

isByte

Use: `b = isByte(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Byte.

isChannelClosed

Use: `b = isChannelClosed(c);`

Pre: `c` is a channel.

Post: `b` now contains true iff `c` is closed.

isChannelEOF

Use: `b = isChannelEOF(x);`

Pre: `x` is a any value.

Post: `b` now contains true iff `x` is the channel EOF value.

isChar

Use: `b = isChar(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Character.

isDouble

Use: `b = isDouble(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Double.

isHashMap

Use: `b = isHashMap(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Map object.

isInstanceOf

Use: `b = isInstanceOf(c,x);`

Pre: `x` is any value, `c` is a string that is a fully qualified class name.

Post: `b` now contains true iff `x` is an instance of the class denoted by `c`.

isInteger

Use: `b = isInteger(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is an Integer object.

isLong

Use: `b = isLong(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Long object.

isPair

Use: `b = isPair(x);`

Pre: `x` can be any value.

Post: `b` contains **`true`** if and only if `x` is a pair (see also the binary operator ‘`:`’).

isPromise

Use: `b = isPromise(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Promise object.

isShort

Use: `b = isShort(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Short object.

isString

Use: `b = isString(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a String object.

isThunk

Use: `b = isThunk(x);`

Pre: `x` is any value.

Post: `b` now contains true iff `x` is a Thunk object.

killFiber

Use: `killFiber();`

Post: The current fiber has been killed so there is no return from the call.

killMachine

Use: `killMachine();`

Post: The current machine has been killed so there is no return from the call.

killTask

Use: `killTask();`

Post: The current task has been killed so there is no return from the call.

listToStream

Use: `y = listToStream(x);`

Pre: `x` must be a list, i.e. either [] (null) or a pair containing a tail that is a list.

Post: `y` now contains a stream that contains the same values as `x` in the same order.

log

Use: `y = log(x);`

Pre: `x` should be a positive Double.

Post: `y` is the natural logarithm of `x`.

log10

Use: `y = log(x);`

Pre: `x` should be a positive Double.

Post: `y` is the base 10 logarithm of `x`.

log1p

Use: `y = log1p(x);`

Pre: `x` should be a Double greater than -1.

Post: `y` is the natural logarithm of `x+1`.

long

Use: `y = long(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same value (if possible), but as a Long object.

makeArray

Use: `a = makeArray(n);`

Pre: `n` must be a non-negative BigInteger, Long, Integer, Short, Character or Byte.

Post: `a` now contains a new Object array of size `n`.

makeChannel

Use: `c = makeChannel();`

Post: `c` now contains a new channel (an is.hi.cs.morpho.Channel object).

makeChannelSelector

Use: `c = makeChannelSelector();`

Post: `c` now contains a new channel selector object, which is a Morpho object that facilitates multiplexing of channels. A channel selector has operations for adding subscriber channels and for starting multiplexing. To use a channel selector to monitor multiple channels for reading from them, you do the following:

1. Create the channel selector.

2. Add a read subscription to the channel selector for each of the channels you wish to monitor.
3. Start the channel selector, which gives you a channel to read from.
4. Read from the channel, which gives you a sequence of channels that are readable.

Similarly, to use a channel selector to monitor multiple channels for writing to them, you do the following:

1. Create the channel selector.
2. Add a write subscription to the channel selector for each of the channels you wish to monitor.
3. Start the channel selector, which gives you a channel to read from.
4. Read from the channel, which gives you a sequence of channels that are writable.

The channel selector has the following messages:

addReadSubscription

Usage: `sel.addReadSubscription(c);`

Pre: `sel` is a channel selector that has not been started and `c` is a channel.

Post: The channel selector now has a read subscription to the channel.

addWriteSubscription

Usage: `sel.addWriteSubscription(c);`

Pre: `sel` is a channel selector that has not been started and `c` is a channel.

Post: The channel selector now has a write subscription to the channel.

start

Usage: `c = sel.start();`

Pre: `sel` is a channel selector that has not been started.

Post: `c` refers to a newly created channel that will receive a sequence of channels. Each channel sent to `c` is either a channel that the channel selector has a read subscription to and is readable or a channel that the selector has a write subscription to and is writable.

makeFiber

Use: `f = makeFiber(c);`

Pre: `c` must be a function (a closure) that takes no argument.

Post: `f` now contains a new continuation that corresponds to a fiber whose execution consists of calling the function `c` and then terminating. The fiber is not running but can be started by using it as the second argument in a call to the BASIS function `transfer`, or by calling the BASIS function `makeReady` with `f` as argument.

makeFiber

Use: `f = makeFiber(c,x);`

Pre: `c` must be a function (a closure) that takes one argument. `x` must be a valid argument to that function.

Post: `f` now contains a new continuation that corresponds to a fiber whose execution consists of calling the function `c` with argument `x` and then terminating. The fiber is not running but can be started by using it as the second argument in a call to the BASIS function `transfer`, or by calling the BASIS function `makeReady` with `f` as argument.

makeFiberReady

Use: `makeFiberReady(f);`

Pre: `f` is a continuation.

Post: Same as for `makeFiberReadyWithValue(f,null)`.

makeFiberReadyWithValue

Use: `makeFiberReadyWithValue(f,x);`

Pre: `f` is a continuation, `x` is any value.

Post: A new fiber has been created in the current task. The fiber will execute the continuation `f` with the value `x`.

makeHashMap

Use: `a = makeHashMap();`

Post: `a` now contains a new empty ConcurrentHashMap Object.

makeMutex

Use: `m = makeMutex();`

Post: `m` now contains a new mutex. No sequence of execution has possession of the mutex.

makePromise

Use: `p = makePromise(c);`

Pre: `c` is a closure that takes no argument.

Post: `p` now contains a new promise that when forced will call the closure to generate the value.

makeReady

Use: `makeReady(f);`

Pre: `f` is a continuation.

Post: Same as for `makeFiberReadyWithValue(f,null)`.

makeReadyWithValue

Use: `makeReadyWithValue(f,x);`

Pre: `f` is a continuation, `x` is any value.

Post: Same as for `makeFiberReadyWithValue(f,x)`.

makeThunk

Use: `t = makeThunk(getter,setter);`

Pre: `getter` and `setter` are both closures. `getter` takes no argument while `setter` takes one.

Post: `t` refers to a thunk that calls `setter` to set a value and `getter` to get a value.

Example: The following code will write two lines, "y=A" and "y=B".

```
1         rec fun test(@x)
2         {
3                 x;
4                 x = "B";
5                 x;
6         };
7         var y = "A";
8         test(
9                 makeThunk(
10                        fun(){writeln("y="++y);y},
11                        fun(v){y=v}
12                )
13        );
```

max

Use: `z = max(x,y);`

Pre: `x` and `y` should be Double values.

Post: `z` is the maximum of the two values.

min

Use: `z = min(x,y);`

Pre: `x` and `y` should be Double values.

Post: `z` is the minimum of the two values.

nanoTime

Use: `t = nanoTime();`

Post: `t` contains a Long value that is a count of nanoseconds from some starting time, same as the static function nanoTime in the Java class java.lang.System.

printExceptionTrace

Use: `printExceptionTrace(&e);`

Pre: `e` is the address of an exception caught in a **`catch`** part of a try-catch expression.

Post: A trace of the exception has been written to standard output.

Example:

```
1        try
2        {
3                throw "Thrown string";
4        }
5        catch( e )
6        {
7                printExceptionTrace(&e);
8        };
```

random

Use: `x = random(n);`

Pre: `n` is an integer, greater than zero.

Post: `x` is a random integer value, $0 \leq x < n$.

readLine

Use: `x = readLine();`

Post: The next input line has been read from standard input and `x` contains the line as a string value.

releaseMutex

Use: `releaseMutex(m);`

Pre: `m` is a mutex that has been acquired.

Post: The mutex has been released. If some other sequences of execution were waiting to acquire the mutex then one of these has acquired it and can continue execution

scanner

Use: `s = scanner();`

Post: `s` refers to a scanner (java.util.Scanner) that is connected to standard input.

setHead

Use: `setHead(x,y);`

Pre: `x` is a pair, `y` can be any value.

Post: The head of the pair `x` has been set to `y`.

setOutputChannel

Use: `setOutputChannel(c);`

Pre: `c` is a writable channel.

Post: The channel has been set as the output channel, this means that all values written with `write()` and `writeln()` are sent to this channel.

setTail

Use: `setTail(x,y);`

Pre: `x` is a pair, `y` can be any value.

Post: The tail of the pair `x` has been set to `y`.

short

Use: `y = short(x);`

Pre: `x` must be BigInteger, Long, Integer, Short, Character or Byte.

Post: `y` now contains the same value (if possible), but as a Short object.

sin

Use: `y = sin(r);`

Pre: `r` is a Double value denoting an angle in radians.

Post: `y` now contains the sine of `r`.

sleep

Use: `sleep(d);`

Pre: `d` is either a Double value denoting an interval in seconds or an integer value denoting an interval in nanoseconds.

Post: The current fiber has slept for the given interval.

loadClassFromFile

Use: `loadClassFromFile(url,name);`

Pre: `url` is the URL of a JAR file, `name` is the fully qualified name of a class in the JAR file.

Post: The class has been loaded.

sqlGetDriver

Use: `driver = sqlGetDriver(classname);`

Pre: `classname` is the fully qualified name of a JDBC driver class that can be found with the current class loader.

Post: The driver has been loaded and `driver` refers to it. This is an object of type java.sql.Driver.

sqlGetDriver

Use: `driver = sqlGetDriver(url,classname);`

Pre: `url` is a string that refers to a ZIP file or path where Java classes can be found, `classname` is the fully qualified name of a JDBC driver class that can be found through the given url.

Post: The driver has been loaded and `driver` refers to it. This is an object of type java.sql.Driver.

sqlGetConnection

Use: `conn = sqlGetConnection(url);`

Pre: `url` is the URL of a JDBC database whose driver has been loaded and registered with the JDBC system.

Post: `conn` refers to a new JDBC database connection to the database.

Note: This call is equivalent to the call `conn = sqlGetConnection(url,null);`

sqlGetConnection

Use: `conn = sqlGetConnection(url,props);`

Pre: `url` is the URL of a JDBC database whose driver has been loaded and registered with the JDBC system. `props` is either null, a list of pairs `[[name,value],...]` or an object of type java.util.Properties.

Post: `conn` refers to a new JDBC database connection to the database. The connection was established using connection parameters from `props`. If `props` is null then an empty java.utils.Properties object was used. If `props` is an object of type java.util.Properties then that was used. If `props` is a list of pairs then a corresponding java.util.Properties object was used.

Note: This function uses the static method `getConnection` in the class java.sql.DriverManager.

sqlGetConnectionFromDriver

Use: `conn = sqlGetConnectionFromDriver(driver,url);`

Pre: `driver` is a JDBC driver, `url` is the URL of a JDBC database that the driver can handle.

Post: `conn` refers to a new JDBC database connection to the database.

Note: This call is equivalent to the call `conn = sqlGetConnectionFromDriver(driver,url,null);`

sqlGetConnectionFromDriver

Use: `conn = sqlGetConnectionFromDriver(driver,url,props);`

Pre: `driver` is a JDBC driver (an object of type java.sql.Driver), `url` is the URL of a JDBC database that the driver can handle. `props` is either null, a list of pairs `[[name,value],...]` or an object of type java.util.Properties.

Post: `conn` refers to a new JDBC database connection to the database. The connection was established using connection parameters from `props`. If `props` is null then an empty java.utils.Properties object was used. If `props` is an object of type java.util.Properties then that was used. If `props` is a list of pairs then a corresponding java.util.Properties object was used.

Note: This function uses the instance method `connect` in the class java.sql.Driver.

sqlCreateStatement

Use: `stmt = sqlCreateStatement(conn);`

Pre: `conn` is a JDBC database connection.

Post: `stmt` refers to a new JDBC statement object for the database, this is an object of type java.sql.Statement.

sqlPrepareStatement

Use: `pstmt = sqlPrepareStatement(conn,stmt);`

Pre: `conn` is a JDBC database connection, `stmt` is a string containing a valid SQL statement.

Post: `pstmt` refers to a new JDBC prepared statement object for the database, this is an object of type java.sql.PreparedStatement.

sqlExecuteUpdate

There are three ways to call this function:

- **Use:** `res = sqlExecuteUpdate(conn,cmd);`

  Pre: `conn` is a JDBC database connection, `cmd` is a string containing a valid SQL update statement, i.e. not a query (SELECT) statement.

  Post: The update has been performed on the database. `res` contains the update count.

- **Use:** `res = sqlExecuteUpdate(pstmt,args);`

  Pre: `pstmt` is a JDBC prepared statement on some database connection, an object of type java.sql.PreparedStatement, `args` is a list of values that are appropriate as arguments in the prepared statement.

Post: The update has been performed on the database with the given arguments. `res` contains the update count.

Note: This call is equivalent to the call:
`res = sqlExecutePreparedUpdate(pstmt,args);`

- **Use:** `res = sqlExecuteUpdate(stmt,cmd);`

  Pre: `stmt` is a JDBC statement on some database connection, an object of type java.sql.Statement, `cmd` is a string that is a valid SQL update command for the underlying database connection.

  Post: The update has been performed on the database. `res` contains the update count.

sqlExecuteQuery

There are three ways to call this function:

- **Use:** `res = sqlExecuteQuery(conn,cmd);`

  Pre: `conn` is a JDBC database connection, `cmd` is a string containing a valid SQL query, i.e. a SELECT statement.

  Post: The query has been performed on the database. `res` contains a java.sql.ResultSet object that is the set of results.

- **Use:** `res = sqlExecuteQuery(pstmt,args);`

  Pre: `pstmt` is a JDBC prepared statement on some database connection, an object of type java.sql.PreparedStatement, `args` is a list of values that are appropriate as arguments in the prepared statement.

  Post: The query has been performed on the database with the given arguments. `res` contains a java.sql.ResultSet object that is the set of results.

  Note: This call is equivalent to the call:
  `res = sqlExecutePreparedQuery(pstmt,args);`

- **Use:** `res = sqlExecuteQuery(stmt,cmd);`

  Pre: `stmt` is a JDBC statement on some database connection, an object of type java.sql.Statement, `cmd` is a string that is a valid SQL query for the underlying database connection.

  Post: The query has been performed on the database. `res` contains a java.sql.ResultSet object that is the set of results.

sqlNext

Use: `res = sqlNext(rs);`

Pre: `rs` is a JDBC result set, i.e. an object of type java.sql.ResultSet. The cursor of the result set should not be positioned after the last row of the result set.

Post: The cursor has been moved one row forward. If it was previously positioned before the first row then it is now positioned on the first row (or after the result set if the set is empty). `res` contains **`true`** if the cursor is positioned on a row, **`false`** if there are no more rows.

sqlCommit

Use: `sqlCommit(conn);`

Pre: `conn` is a JDBC connection, i.e. an object of type java.sql.Connection.

Post: Any pending changes to the database through the connection have been committed to the database and all locks in the database through the connection have been released.

sqlRollback

Use: `sqlRollback(conn);`

Pre: `conn` is a JDBC connection, i.e. an object of type java.sql.Connection.

Post: Any pending changes to the database through the connection have been rolled back and all locks in the database through the connection have been released.

sqlClose

There are three ways to call this function:

- **Use:** `sqlClose(conn);`

  Pre: `conn` is a JDBC database connection.

  Post: The connection has been closed.

- **Use:** `sqlClose(rs);`

  Pre: `rs` is a JDBC result set.

  Post: The result set has been closed.

- **Use:** `sqlClose(stmt);`

  Pre: `stmt` is a JDBC statement (of type java.sql.Statement).

  Post: The statement has been closed.

sqlSetAutoCommit

Use: `sqlSetAutoCommit(conn,flag);`

Pre: `conn` is a JDBC connection, i.e. an object of type java.sql.Connection. `flag` is either **`true`** or **`false`**.

Post: The connections autocommit property has been set to the value of the flag. If the autocommit property is true then each update performed through the connection is implicitly followed by a commit, otherwise not.

sqlGetObject

Use: `res = sqlGetObject(rs,row);`

Pre: `rs` is a JDBC result set whose cursor is positioned on some row. `row` is either a valid row number (the first valid one is 1) or a string that is a valid row name.

Post: `res` contains the value in the row.

sqlExecutePreparedQuery

Use: `res = sqlExecutePreparedQuery(pstmt,args);`

Pre: `pstmt` is a JDBC prepared query statement on some database connection, an object of type java.sql.PreparedStatement, `args` is a list of values that are appropriate as arguments in the prepared statement.

Post: The query has been performed on the database with the given arguments. `res` contains a java.sql.ResultSet object that is the set of results.

Note: This call is equivalent to the call:
`res = sqlExecuteQuery(pstmt,args);`

sqlExecutePreparedUpdate

Use: `res = sqlExecutePreparedUpdate(pstmt,args);`

Pre: `pstmt` is a JDBC prepared update statement on some database connection, an object of type java.sql.PreparedStatement, `args` is a list of values that are appropriate as arguments in the prepared statement.

Post: The update has been performed on the database with the given arguments. `res` contains the update count.

Note: This call is equivalent to the call:
`res = sqlExecuteUpdate(pstmt,args);`

sqlForAll_fromResultSet_do

Use: `sqlForAll(var x) fromResultSet(rs) do{...};`

Pre: `rs` is a JDBC result set whose cursor is not positioned after the last row.

Post: The body `{...}` has been executed once for each remaining row in the result set, excluding the current, if any. Each time the body is executed the variable `x` is given a new value before the execution and that value is the value of the first column in the row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x; sqlForAll_fromResultSet_do(@x,rs,@{...})};`

sqlForAll_fromQuery__do

Use: `sqlForAll(var x) fromQuery(stmt,query) do{...};`

Pre: `stmt` is a JDBC statement (an object of type java.sql.Statement), query is a string containing an SQL SELECT statement.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variable `x` is given a new value before the execution and that value is the value of the first column in the row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:

```
{var x; sqlForAll_fromQueryi__do(@x,stmt,query,@{...})};
```

sqlForAllRows_fromResultSet_do

Use: `sqlForAllRows(var x) fromResultSet(rs) do{...};`

Pre: `rs` is a JDBC result set whose cursor is not positioned after the last row.

Post: The body `{...}` has been executed once for each remaining row in the result set, excluding the current, if any. Each time the body is executed the variable `x` is given a new value before the execution and that value is an array containing all the values of the row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:

```
{var x; sqlForAllRows_fromResultSet_do(@x,rs,@{...})};
```

sqlForAllRows_fromQuery__do

Use: `sqlForAllRows(var x) fromQuery(stmt,query) do{...};`

Pre: `stmt` is a JDBC statement (an object of type java.sql.Statement), query is a string containing an SQL SELECT statement.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variable `x` is given a new value before the execution and that value is an array containing all the values in the row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:

```
{var x; sqlForAllRows_fromQueryi__do(@x,stmt,query,@{...})};
```

sqlForAllRows_fromPreparedQuery__do

Use: `sqlForAllRows(var x) fromPreparedQuery(pstmt,args) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses zero or more arguments, args is a list of values that are valid arguments for the query.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query with the given arguments. Each time the body is executed the variable `x` is given a new value before the execution and that value is an array containing all the values in the row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x; sqlForAllRows_fromPreparedQueryi__do(@x,pstmt,args,@{...})};`

sqlForAll_fromPreparedQuery_do

Use: `sqlForAll(var x) fromPreparedQuery(pstmt) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses no argument.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variable `x` is given a new value before the execution and that value is the value from the first column in the result row.

Note: The variable `x` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x; sqlForAll_fromPreparedQueryi_do(@x,pstmt,@{...})};`

sqlForAll__fromPreparedQuery__do

Use: `sqlForAll(var x, var y) fromPreparedQuery(pstmt,args) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses zero or more arguments and returns rows with at least two columns, args is a list of values that are valid arguments for the query.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query with the given arguments. Each time the body is executed the variables `x` and `y` are given new values before the execution and those values are the values in the first and second columns, respectively.

Note: The variables `x`, `y` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y; sqlForAll__fromPreparedQuery__do(@x,@y,pstmt,args,@{...})};`

sqlForAll__fromPreparedQuery_do

Use: `sqlForAll(var x, var y) fromPreparedQuery(pstmt) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses no argument and returns rows with at least two columns.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variables `x` and `y` are given new values before the execution and those values are the values in the first and second columns, respectively.

Note: The variables `x`, `y` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y; sqlForAll__fromPreparedQuery_do(@x,@y,pstmt,@{...})};`

sqlForAll___fromPreparedQuery__do

Use: `sqlForAll(var x, var y, var z) fromPreparedQuery(pstmt,args) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses zero or more arguments and returns rows with at least three columns, args is a list of values that are valid arguments for the query.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query with the given arguments. Each time the body is executed the variables `x`, `y` and `z` are given new values before the execution and those values are the values in the first, second and third columns, respectively.

Note: The variables `x`, `y` and `y` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y,z; sqlForAll___fromPreparedQuery__do(@x,@y,@z,pstmt,args,@{...})};`

sqlForAll____fromPreparedQuery__do

Use: `sqlForAll(var x, var y, var z,var t) fromPreparedQuery(pstmt,args) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses zero or more arguments and returns rows with at least four columns, args is a list of values that are valid arguments for the query.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query with the given arguments. Each time the body is executed the variables `x`, `y`, `z` and `t` are given new values before the execution and those values are the values in the first, second, third and fourth columns, respectively.

Note: The variables `x`, `y`, `z` and `t` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y,z,t; sqlForAll____fromPreparedQuery__do(@x,@y,@z,@t,pstmt,args,@{.`

sqlForAll____fromPreparedQuery_do

Use: `sqlForAll(var x, var y, var z, var t) fromPreparedQuery(pstmt) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses no argument and returns rows with at least four columns.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variables `x`, `y`, `z` and `t` are given new values before the execution and those values are the values in the first, second, third and fourth columns, respectively.

Note: The variables `x`, `y`, `z` and `t` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y,z,t; sqlForAll____fromPreparedQuery_do(@x,@y,@z,@t,pstmt,@{...})};`

sqlForAll___fromPreparedQuery_do

Use: `sqlForAll(var x, var y, var z) fromPreparedQuery(pstmt) do{...};`

Pre: `pstmt` is a JDBC prepared statement (an object of type java.sql.PreparedStatement) that uses no argument and returns rows with at least three columns.

Post: The body `{...}` has been executed once for each row in the result set that is generated from the query. Each time the body is executed the variables `x`, `y` and `z` are given new values before the execution and those values are the values in the first, second and third columns, respectively.

Note: The variables `x`, `y` and `z` and the body `{...}` are by-name parameters in the call to the function. The call above can also be written as:
`{var x,y,z; sqlForAll___fromPreparedQuery_do(@x,@y,@z,pstmt,@{...})};`

sqrt

Use: `y = sqrt(x);`

Pre: `x` is a non-negative Double value.

Post: `y` now contains the square root of `x`.

startBackgroundTask

Use: startBackgroundTask(f);|

Pre: `f` must be a function (a closure) taking no argument.

Post: A new background task has been started in a special background machine that has multiple threads. The task executes the given function and then terminates.

startBackgroundTask

Use: startBackgroundTask(f,x);|

Pre: `f` must be a function (a closure) taking one argument, `x` must be a valid argument to the function.

Post: A new background task has been started in a special background machine that has multiple threads. The task executes the given function with the given argument and then terminates.

startFiber

Use: startFiber(f);|

Pre: `f` must be a function (a closure) taking no argument.

Post: A new fiber has been started in the current task. The fiber executes the given function and then terminates.

startFiber

Use: startFiber(f,x);|

Pre: `f` must be a function (a closure) taking one argument, `x` must be a valid argument to the function.

Post: A new fiber has been started in the current task. The fiber executes the given function with the given argument and then terminates.

startForegroundTask

Use: startForegroundTask(f);|

Pre: `f` must be a function (a closure) taking no argument.

Post: A new foreground task has been started in the special foreground machine that has a single thread. The task executes the given function and then terminates.

startForegroundTask

Use: startForegroundTask(f,x);|

Pre: `f` must be a function (a closure) taking one argument, `x` must be a valid argument to the function.

Post: A new foreground task has been started in the special foreground machine that has a single thread. The task executes the given function with the given argument and then terminates.

startMachine

Use: startMachine(f,n);|

Pre: `f` must be a function (a closure) taking no argument. n is an integer.

Post: A new machine has been started. The machine executes the given function and then terminates. The number of threads used in the machine is controlled by n. If n is zero or negative then the machine will be allocated a number of threads, exceeding the number cores in the computer. Otherwise the machine will be allocated n threads.

startMachine

Use: startMachine(f,n,x);|

Pre: `f` must be a function (a closure) taking one argument, `x` must be a valid argument to the function. n is an integer.

Post: A new machine has been started. The machine executes the given function with the given argument and then terminates. The integer argument functions as described above in the other variant of startMachine.

startTask

Use: startTask(f);|

Pre: `f` must be a function (a closure) taking no argument.

Post: A new task has been started in the current machine. The task executes the given function and then terminates.

startTask

Use: startTask(f,x);|

Pre: `f` must be a function (a closure) taking one argument, `x` must be a valid argument to the function.

Post: A new task has been started in the current machine. The task executes the given function with the given argument and then terminates.

streamAppend

Use: `z = streamAppend(x,y);`

Pre: `x` and `y` are streams.

Post: `z` is a stream that contains the values in `x` followed by the values in `y`.

streamConcat

Use: `y = streamConcat(x);`

Pre: `x` is a stream of streams.

Post: `y` is the stream of values resulting when all the streams in `x` are concatenated.

streamConcatMap

Use: `y = streamConcatMap(f,x);`

Pre: `x` is a stream, `f` is a function of one argument that returns a stream.

Post: `y` is the stream of values resulting from concatenating the streams resulting from applying `f` to all the the values in `x` in sequence. See also >>=.

streamHead

Use: `y = streamHead(x);`

Pre: `x` is a non-empty stream.

Post: `y` is the head of the stream `x`.

streamMap

Use: `y = streamMap(f,x);`

Pre: `x` is a stream, `f` is a function of one argument.

Post: `y` is the stream of values resulting when `f` is applied to all the values in `x`.

streamTail

Use: `y = streamTail(x);`

Pre: `x` must be a non-empty stream.

Post: `y` is the head of `y`.

Example: The code

```
1 {
2   rec val e = #[ 1 $ e ];
3   writeln(streamHead(streamTail(e)));
4 }
```

will write the number 1.

streamToList

Use: `y = streamToList(x);`

Pre: `x` is a finite stream.

Post: `y` is the list of values in `x`.

streamToList

Use: `y = streamToList(x,n);`

Pre: `n` is an integer ≥ 0, `x` is a stream.

Post: `y` is the list of the first `n` values in `x`, or a list containing all the values in `x` if `x` contains less than `n` values.

tail

Use: `y = tail(x);`

Pre: `x` is a pair.

Post: `y` is the tail of `x`.

take

Use: `y = take(n,x);`

Pre: `n` is a non-negative integer, `x` is a stream of length at least `n`.

Post: `y` is a new list containing the first `n` values of `x`.

tan

Use: `y = tan(r);`

Pre: `r` is a Double denoting an angle in radians.

Post: `y` is the tangent of `r`.

transfer

Use: `y = transfer(from,to,x);`

Pre: The last argument, `x`, can be any value. There are three possible valid cases for the other arguments:

1. `from` may be null and `to` may be a continuation (an is.hi.cs.morpho.Continuation object) that has previously received initialization by being referred to by a first argument in a call to `transfer`.
2. `from` may be a reference to a variable and `to` may be null.
3. Neither `from` nor `to` may be null, in which case `from` must be a reference to a variable and `to` must be refer to a continuation.

Post: The results for the three different cases are:

1. The current fiber is suspended and control passes to the continuation denoted by `to`. The continuation denoted by `to` will resume from the previously executed call to `transfer` in that fiber, if any, and that call will return `x` as it's value. However, the continuation may have been created in other ways, such as by calling `makeFiber` or `forkSuspendedFiber`. In that case the third parameter, `x` is sometimes effectively ignored.
2. The current fiber continues just as if nothing had happened, and `y` now contains `x`. However, `*from` now contains a continuation that allows a later transfer to the exact same point of control, using a later call to `transfer`.
3. In the case where neither `from` nor `to` are null, the current fiber is suspended, but it's state is saved in `*from`, so it can be resumed later using a later call to `tranfer`. Control passes to the continuation denoted by `to`, to the call to `transfer` that initialized `to`. The value returned by that call is `x`. As in the first case above, however, the continuation referred to by `to` may have been created by other means than be calling `transfer`, in which case `x` is sometimes ignored.

Example: The following code segment will will execute two transfer calls. The first one will return **`false`** and cause the second one to execute, which will return to the same **`if`**-condition as the first one, but in that case the condition will be **`true`**, and therefore a line contining `"Hello world"` will be written.

```
1         var f;
2         if( transfer(&f,null,false) )
3         {
4                 writeln("Hello world");
5         }
6         else
7         {
8                 transfer(null,f,true);
9         };
```

Note 1: This is a very powerful and primitive function that can often be used when other methods of flow control, including the exception handling functionality, are not flexible enough.

Note 2: A continuation can be created in one task and then resumed later in another task.

turtle

Use: `t = turtle();`

Post: A new graphics window has been created and made visible on the screen. The value `t` refers to a new turtle object that can be used to draw upon the graphics window. Originally the graphics window has a coordinate system with x and y ranging from -1.0 to 1.0, with the bottom-left corner at $(-1, -1)$. The turtle is originally located at $(0, 0)$ with a direction to the left, i.e. an angle of 0 radians. The turtle referred to by `t` responds to the following messages:

forward

Use: `t.forward(d);`

Pre: `t` is a turtle, `d` is a Double value.

Post: The turtle has been moved a distance of `d` in its current direction, and if it is down then a line has been drawn with the current color from the old position to the new position.

left

Use: `t.left();`

Pre: `t` is a turtle.

Post: The turtle has been turned 90 degrees (*pi*/2 radians) to the left.

right

Use: `t.right();`

Pre: `t` is a turtle.

Post: The turtle has been turned 90 degrees (*pi*/2 radians) to the right.

turn

Use: `t.turn(a);`

Pre: `t` is a turtle, `a` is a Double value.

Post: The turtle has been turned `a` radians to the left.

color

Use: `t.color(r,g,b);`

Pre: `t` is a turtle, `r`, `g`, and `b` are integer values, $0 \leq$ `r`, `g`, `b` < 256.

Post: The color of the turtle has been set to the color denoted by the integer values, where `r`, `g`, and `b` stand for red, green and blue intensities, respectively.

hide

Use: `t.hide();`

Pre: `t` is a turtle.

Post: The turtle has been hidden and will not be visible on the graphics window.

show

Use: `t.show();`

Pre: `t` is a turtle.

Post: The turtle has been shown and will be visible on the graphics window.

up

Use: `t.up();`

Pre: `t` is a turtle.

Post: The pen of the turtle has been lifted and will not draw on the graphics window.

down

Use: `t.down();`

Pre: `t` is a turtle.

Post: The pen of the turtle has been put down and will draw on the graphics window.

set direction

Use: `t.direction(a);`

Pre: `t` is a turtle, `a` is a Double value.

Post: The turtle has been turned so that it points `a` radians to the left of the x-axis.

moveTo

Use: `t.moveTo(x,y);`

Pre: `t` is a turtle, `x` and `y` are Double values.

Post: The turtle has been moved to (`x`,`y`) without drawing anything.

drawTo

Use: `t.moveTo(x,y);`

Pre: `t` is a turtle, `x` and `y` are Double values.

Post: The turtle has been moved to (`x`,`y`) and has drawn a line from the old position to the new one using the current color.

x position

Use: `z = t.x;`

Pre: `t` is a turtle.

Post: `z` contains the x-position of the turtle.

y position

Use: `z = t.y;`

Pre: `t` is a turtle.

Post: `z` contains the y-position of the turtle.

angle

Use: `a = t.angle;`

Pre: `t` is a turtle.

Post: `a` contains the angle of the turtle.

hidden

Use: `h = t.hidden;`

Pre: `t` is a turtle.

Post: `h` contains true iff the turtle is hidden.

up

Use: `u = t.up;`

Pre: `t` is a turtle.

Post: `u` contains true iff the turtle's pen is lifted.

Set Scaling

Use: `t.setScaling(xorg,yorg,xscale,yscale);`

Pre: `t` is a turtle, `xorg`, `yorg`, `xscale`, and `yscale` are Double values.

Post: The scaling transformation of the turtle has been set to the given values. `xorg` is the x coordinate in the graphics window that the point (0,0) maps to, `yorg` is the y coordinate in the graphics window that the point (0,0) maps to, `xscale` is the scale multiplier in the x-direction, and `yscale` is the scale multiplier in the x-direction. This means in general that a turtle coordinate (`x`,`y`) will map to the coordinate (`xorg`+`x`*`xscale`,`yorg`+`y`*`yscale`).

The original scaling of the turtle for a newly created graphics window is

- `xorg` = 250.0
- `yorg` = 250.0
- `xscale` = 250.0
- `yscale` = -250.0

x origin

Use: `xorg = t.xorg`

Pre: `t` is a turtle.

Post: xorg is the x origin of the turtle, see the setScaling message II on the preceding page.

y origin

Use: `yorg = t.yorg`

Pre: `t` is a turtle.

Post: yorg is the y origin of the turtle, see the setScaling message II on the previous page.

x scale

Use: `xscale = t.xscale`

Pre: `t` is a turtle.

Post: xscale is the x scale of the turtle, see the setScaling message II on the preceding page.

y scale

Use: `yscale = t.yscale`

Pre: `t` is a turtle.

Post: yscale is the y scale of the turtle, see the setScaling message II on the previous page.

width

Use: `w = t.width`

Pre: `t` is a turtle.

Post: w is the width of the content area of the graphics window containing the turtle.

height

Use: `h = t.width`

Pre: `t` is a turtle.

Post: h is the height of the content area of the graphics window containing the turtle.

setSize

Use: `t.setSize(w,h)`

Pre: `t` is a turtle, `w` and `h` are Double values.

Post: The width and height of the content area of the graphics window containing the turtle have been set to the values given.

window

Use: `w = t.window`

Pre: `t` is a turtle.

Post: w refers to the Java object that stands for the graphics window containing the turtle.

with

Use: `res = with(@x,@{...});`

Post: The body `{...}` has been executed once and `res` contains the value returned by the body.

Examples: This will write "Hello World":

```
with(val x="Hello World"){writeln(x)};
```

This will write "HelloHello World":

```
writeln(with(val x="Hello"){x++x++" World"});
```

write

Use: `write(x);`

Pre: `x` is any value.

Post: A text representation of `x` has been written to standard output.

writeln

Use: `writeln();`

Post: A newline has been written to standard output.

writeln

Use: `writeln(x);`

Pre: `x` is any value.

Post: A text representation of `x` and a subsequent newline has been written to standard output.

yieldFiber

Use: `yieldFiber();`

Post: The current fiber has yielded control to other fibers in the current task and has then subsequently received control again.

Note: Only one fiber in each task can be running at any instance of time.

yieldTask

Use: `yieldTask();`

Post: The current task has yielded control to other tasks in the current machine and has then subsequently received control again.

Note: Multiple tasks can run concurrently in the same machine.

Part III.

Index

Index