

Turtle

Reference Manual
Version 1.0.0

Martin Grabmueller

Copyright © 2003 Martin Grabmueller

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

1 Introduction

Turtle is both a programming language and the name of a compiler for this language.

The language Turtle is a programming language which supports constraint imperative programming, that means that standard imperative programming constructs can be combined with the declarative style of constraint programming.

The chapter after this introduction describes the Turtle compiler from the user's point of view, you will learn how to run the compiler and how to develop programs using it.

After that, in the third chapter, the Turtle programming language is presented in detail, including formal descriptions like EBNF grammars, and an informal description of the semantics of Turtle.

The fourth chapter contains the reference documentation for the standard library modules which come with the Turtle system.

The manual is closed by a a glossary and an index of all functions and variables of the standard library.

1.1 About Turtle

The Turtle programming language has these main characteristics:

Imperative

Turtle has the basic imperative constructs like variables, side-effects, loops, procedures, procedure calls, statements, and statement sequencing. Thus the semantics of a Turtle program relies on state and time.

Constraints

Turtle supports constraint programming. That means that constraints can be asserted on program variables, so that these constraints can be checked and values for the variables can be deduced.

Higher-order programming

Functions and procedures are first-class values, they can be returned from functions and even be stored in data structures. Additionally to the imperative and constraint programming styles, this makes functional programming possible.

Garbage collection

Memory is managed automatically in Turtle programs. Garbage collection is now standard in new programming languages, since it frees the programmer from many possible bug sources, such as dangling pointers (because references to freed memory regions are used) or memory leaks (because unneeded references are kept).

Module system

Turtle comes with a module system for structuring programs in smaller, encapsulated subsystems. This is a key feature for software engineering, because it makes the interfaces between subsystems clean and makes life easier if several programmers are working on different parts of a larger program.

Additionally, modules can be parametrized over data types. The programmer can thus write generic modules which implement abstract data types which can be used with arbitrary objects of arbitrary types.

Rich set of data types

Turtle comes with builtin basic data types (integers, reals, booleans, strings, characters) as well as aggregate types such as arrays, lists and tuples.

Efficiency Turtle programs run reasonably fast, mainly because they are not interpreted but compiled to machine code (via generating C code and running a C compiler).

1.2 Turtle History

Turtle is the product of my master's thesis, written at the Technical University of Berlin.

The language definition was sketched in February 2002, and after the thesis was officially started, development towards turning it into a usable constraint imperative language began.

1.3 Turtle Future

I don't know.

2 Using Turtle

This chapter describes how to use the Turtle compiler. You will learn how to invoke the compiler in order to produce an executable program, what command line options the compiler understands and how the compiler can be used to access the underlying system and how to produce documentation from source code comments.

2.1 Compiling source programs

The Turtle compiler is run by executing the command `turtle`, giving the names of the source files to be compiled on the command line:

```
turtle ex1.t
```

Turtle source code files are named with the file name extension `.t` by convention.

The compiler will generate a bunch of files from each Turtle source file:

- '`ex1.ifc`' This is the interface file in which the exported identifiers, imported modules and some other data is stored. Interface files are read when modules are imported.
- '`ex1.c`' This is the C source code generated for the input file. It is compiled by the C compiler to produce the file '`ex1.o`' mentioned below.
- '`ex1.h`' This is the header file (for the C compiler) which contains declarations for all exported variables and functions and for the initialization files.
- '`ex1.o`' This is the object code for the input file, it contains the machine code.

If you don't want to compile a library module, but a complete program, you have to give the `--main=name` option to the compiler. Section 2.2 [Command line options], page 3 for details.

2.2 Command line options

The following command line options are available.

- `-h, --help`
This option will force the compiler to show a short usage message and to exit successfully.
- `-v, --version`
The compiler will print its version number and exit successfully.
- `-m, --main=name`
This option must be given when compiling the main module of a program. It tells the compiler to link the module with all imported modules and to produce an executable program called *name*. With this option, only one source file may be given, which must be the main module.
- `-p, --module-path=path`
Tells the compiler to look in all directories in *path* (which must be a colon-separated list of directory names) when searching for imported modules.
- `-d, --debug=MODIFIER`
This options switches on one or more debugging options. Each debugging option is denoted by a different letter; they must be given one after the other without intervening spaces, as in the following example:

```
turtle --debug=ae
```

 - `a` This option causes the compiler to print out the abstract syntax tree of each parsed source file after reading it.

- e With this option, the compiler prints the top-level environment of each source file after parsing and analyzing it.
 - i The highlevel intermediate language (HIL) representation of the source program will be printed to standard out just after type checking.
 - b The lowlevel intermediate language (LIL) representation of the source module will be printed to standard out just after code generation.
- z, --pragma=*pragma***
This option is used to pass additional information about the source file or the compilation process. The following pragmas can be given:
- handcoded**
Handcoded modules (that is, modules which contain functions implemented in C) must be compiled with this option, otherwise the compiler will refuse to compile the source file. This is a safety feature, because declarations for handcoded functions can be easily written by error and would otherwise lead to hard understandable error messages.
 - compile-only**
With this option, the C compiler will not be run on the generated C code. This option is mainly useful for debugging the compiler.
 - static** Causes the linker to statically link the program. Only has an effect if the **--main=*name*** option is also given. This option may require GCC.
 - turtledoc**
Instead of compiling the source files on the command line, the compiler will extract documentation comments and create Texinfo documentation for the modules. For each module to be compiled, a Texinfo node will be written to standard output with documentation for all data types, variables and functions defined in the module.
 - deps** In addition to normal compilation, a the dependencies of the source file will be written to a file with the same basename as the input file, but with the extension **‘.P’**. The dependencies are formatted to be usable with **‘make’**.
 - deps-stdout**
Like the pragma **deps**, but instead of writing to a dependency file, the dependencies are written to standard output.
- O, --optimize=FLAGS**
Set some flags for the Turtle optimizer and the C compiler. By default all Turtle optimizations are on and the C optimizations are off. One or more of the following flags can be given:
- C** Optimize module-local calls.
 - c** Do not optimize module-local calls.
 - J** Convert module-local calls to jumps.
 - j** Do not convert module-local calls to jumps.
 - G** Merge all GC checks which appear in a basic block into one at the start of the block.
 - g** Do not merge all GC checks in one basic block.
 - 0...6** Set the optimization level for the C compiler to the given value. This option may require GCC.

2.3 Handcoding

A lot of low-level functions cannot be implemented in Turtle, because there is currently no way to access operating system features or the C library directly. Therefore, ‘handcoded’ modules have been added to the Turtle compiler.

Handcoded modules can contain normal Turtle definitions for functions, variables and data types, but additionally they can contain functions whose body has been omitted, and whose implementation is written in C in another place. This other place is a C source code file which must be named like the module to be compiled, but with the added extension ‘.i’. That means that the implementation of handcoded functions in the module file ‘core.t’ must reside in the file called ‘core.t.i’.

Currently, there are two kinds of handcoded functions. The first is to simply omit the function body and give the implementation as a C macro which will be placed into the generated C code by the compiler. This has the advantage that there is no runtime penalty for using a handcoded function, and that the macro can directly access all virtual registers without the need to flush them to their memory locations. The disadvantage is that it is quite easy to mess up the virtual machine’s state. Handcoded functions written as macros are called “implementation macros”.

The second method is to write down the function header, followed by an equal sign and a string which names a C function. This function will then be called from the generated code. The parameters to the declared Turtle function are passed as normal C parameters to the named function, so that there is no way to mess with the Turtle stack or registers. This method is recommended and should be used unless there is an urgent need for efficiency or access to the virtual machine from inside the handcoded function. Functions of this kind are called “mapped functions”, because their functionality is mapped directly to a C function.

2.3.1 Compiling handcoded modules

Because it is easy to break your programs with handcoded modules, the compiler will only accept such modules when explicitly instructed with the pragma option `handcoded`. For example:

```
turtle --pragma=handcoded math.t
```

2.3.2 Implementation include file

An implementation include file must list all needed implementation macros and mapped function implementations. If the macros or functions need any declaration from the C library, file inclusion statements can also appear in the file, and C variables and typedefs can also be written here.

2.3.3 Implementation macros

These macro definitions must be written in a file called like the Turtle source file, but with a ‘.i’ added to the file name. This file is included into the compiled C file. The implementation macros are expected to be named like the mangled function names, appended with the string `_implementation`, and they must not have parameters. The easiest way to get at the mangled name is to implement the function in the Turtle module, compile it with the `--pragma=handcoded` option, ignore the errors from the linker and take the mangled name from the C output file. Additionally, you may want to add the function’s signature to the implementation macro, for documentation purposes.

Function entry and exit code are compiled like for normal Turtle functions, so that the parameters to the Turtle function can be found in the first elements of the array `env->locals`.

This is an example for an implementation macro as used in the library module `core`. The following is code from the module file ‘core.t’:

```
fun version (): string;
```

And this is the corresponding implementation macro from the file ‘core.t.i’:

```
#define core_version_F_0__To_S_implementation          \
{                                                       \
    TTL_SAVE_REGISTERS;                                \
    {                                                   \
        ttl_global_acc = ttl_string_to_value (ttl_version_string (), \
                                              -1);        \
    }                                                 \
    TTL_RESTORE_REGISTERS;                            \
}
```

There are some important rules to keep in mind when writing handcoded functions, but note that there may be other problems as well, which have just not yet come to the surface:

- They must expand into proper C compound statements (because the macro call is not terminated by a semicolon).
- They must not expect parameters.
- They must be called like the mangled name of the function they implement, with the string `_implementation` appended.
- When they call any runtime function which might invoke the garbage collector, the code must be surrounded by a `TTL_SAVE/RESTORE_REGISTERS` pair and operate on global registers
- The parameters to the functions must be taken from `env->locals[X]`, where `X` ranges from 0 to the number of parameters - 1.
- No higher-order programming is allowed.
- Implementation macros may not call back to Turtle code, since there is no way to set up continuation labels, so the Turtle code would not be able to return.
- Boxing/unboxing of Turtle values must be done by hand. Be careful not to screw up the tagging of Turtle values, or the program will most probably blow up. Handcoding is your own risk, since the compiler cannot do anything about it.
- You are responsible for checking for null pointers as well when working with compound data types, and for raising the correct exception by hand.

2.3.4 Mapped functions

Mapped functions are much easier to write than implementation macros, because you don’t have to take care of heap overflows causing garbage collection, for extracting the parameters from the correct environment locations or for finding the mangled name of functions from the generated C code. As an example for a mapped functions, we’ll examine the `getlogin` function from the `unix` module.

This is the function declaration as it appears in the Turtle file:

```
fun getlogin (): string = "ttl_getlogin";
```

The function is declared as a mapped function, which should be mapped to the C function `ttl_getlogin`. This function is an additional layer between the code generated for the Turtle function and the C library function. Its responsibility is to unbox the parameters (if any), call the C library function and to box the result and return it to the caller. Here is how this looks like for the `getlogin` example:

```
ttl_value
ttl_getlogin (void)
{
    char * val = getlogin ();
```

```

    if (val)
      return ttl_string_to_value (val, -1);
    else
      return ttl_string_to_value ("", 0);
  }

```

Compared to the implementation macros described above, this is really easy. But note that the same restrictions apply as for the implementation macros described in the previous subsection.

In the future the Turtle compiler could be equipped with the additional functionality to generate such glue code itself, so that all boxing/unboxing etc. would be done by compiler-generated code, on the basis of some data type declarations.

2.4 Documenting Turtle Modules

Documentation is a very important part in software creation. It is not only important for programmers using third-party modules, but also to make sure that one's own modules will be reusable and can be still understood in the future.

The Turtle compiler has support for simple inline documentation of Turtle modules, called *turtledoc*. It is similar in spirit to the well-known ‘JavaDoc’ program which extracts documentation comments from Java source files and creates fully indexed and linked HTML documentation for Java packages. Though *turtledoc* is much simpler, it is nevertheless quite useful and has been used to create the documentation for the Turtle standard library in this manual (see Chapter 5 [Standard library], page 25).

2.4.1 Preparing Turtle Modules for *turtledoc*

The basic concept of *turtledoc* are so-called “documentation strings”. Each function, variable or data type of a module can have a documentation string attached, and even the module itself can have one. The first step in using *turtledoc* is to attach documentation strings to these entities, and the second step is to use the Turtle compiler to extract these strings from the module and have them printed together with the name, type and kind of the entity they document.

This subsection documents the first step, the second step is described in the next subsection.

For illustration, we will prepare an example module with documentation strings. This is the example:

```

module docex;

fun do_nothing ()
end;

fun do_nothing2 ()
end;

fun ignored ()
end;

fun do_nothing_too ()
  var i: int := 10000;
  while i > 0 do
    i := i - 1;
  end;
end;

```

We could already run the compiler on this and instruct it to extract documentation, but the result would be a bit disappointing:

```
@node docex module
@subsection docex module
@cpindex docex (Module)

@deftypefn {Library function} {} do_nothing ()
@end deftypefn

@deftypefn {Library function} {} do_nothing2 ()
@end deftypefn

@deftypefn {Library function} {} ignored ()
@end deftypefn

@deftypefn {Library function} {} do_nothing_too ()
@end deftypefn
```

Turtledoc comments are written like normal comments, but with an asterisk directly after the comment starting characters:

```
/* Like this...
** ... or this. */
```

The documentation strings must precede the entity they document. A module's documentation string will be written out preceding the documentation of the variables, functions etc., Variable and function documentation will be placed between their Texinfo documentation constructs. Datatype documentation is additionally equipped with the definition of the data type, which is normally very instructive for the reader who wishes to use the data type.

A function can be omitted from the output by providing it with a documentation string starting with a - (minus sign).

Often it is useful to group several functions together with the same documentation, for example when they do similar things and only differ in the types of their arguments. This can be accomplished by writing the documentation string down with the first of the functions to be grouped, and by placing the other functions of the group directly behind that, where each of the functions has a documentation string starting with " (doublequote).

This is the example, now equipped with documentation strings:

```
/* Example module for turtledoc.
module docex;

/* This function does nothing.
fun do_nothing ()
end;

/* ""
fun do_nothing2 ()
end;

/* -
fun ignored ()
end;

/* This function does nothing too, but expensively.
```

```

fun do_nothing_too ()
  var i: int := 10000;
  while i > 0 do
    i := i - 1;
  end;
end;

```

And this is the output produced from the annotated source code:

```

@node docex module
@subsection docex module
@cpindex docex (Module)

Example module for turtledoc.

@deftypefn {Library function} {} do_nothing ()
@deftypefnx {Library function} {} do_nothing2 ()
This function does nothing.
@end deftypefn

@deftypefn {Library function} {} do_nothing_too ()
This function does nothing too, but expensively.
@end deftypefn

```

We can see that the module's documentation comes directly after the node header. The `do_nothing` and `do_nothing2` functions are grouped together and the `ignore` function is not documented at all. The last function, `do_nothing_too`, is documented separately.

The documentation strings can contain arbitrary Texinfo markup, such as for tables or node references. The strings are not interpreted or modified in any way.

2.4.2 Extracting the Documentation

After having equipped the source code with properly marked documentation strings, you have to run the compiler on the source file and instruct it to create the `turtledoc` documentation. This is done by simply giving the pragma option `turtledoc` to the compiler, like for the example module from the previous subsection:

```
turtle --pragma=turtledoc docex.t
```

The documentation is then written to standard output and can be included into the Texinfo documentation for your project.

3 Language Reference

This chapter describes the Turtle programming language. First, the syntax and semantics of the language are presented, then the runtime environment and the standard library are documented.

3.1 Turtle Grammar

This section describes the grammar of the Turtle programming language.

3.1.1 Notation

The lexical and grammatical structure of Turtle is described using Extended Backus Naur Form (EBNF). This notation is summarized below.

<code>A ::= E</code>	The non-terminal A produces the form E.
<code>E F</code>	Alternative; either E or F is produced.
<code>{ E }</code>	The form E is repeated, possibly zero times.
<code>[E]</code>	The form E is optional, it may be omitted completely.
<code>(E)</code>	For grouping, denotes E itself.
<code><empty></code>	Denotes the empty string.

3.1.2 Lexical Structure

Turtle programs are composed by tokens of the following token classes:

Reserved words

Reserved words look like identifiers (see below), but they have special meanings in the source program. They are summarized in the Reserved Words Table below. Identifiers denoting reserved words cannot be used for other purposes (variable names, function names, etc.).

Identifiers

An identifier is used for naming semantic entities in the program text: variables, functions, constraints, modules, etc.

Like in most other imperative languages descending from Algol, identifiers start with an alphabetic letter or an underscore, followed by a number of alphabetic letters, underscores or digits. Additionally, identifiers may be terminated with a question mark.

There is no length restriction on identifiers.

```
Ident ::= (Letter | Underscore) {Letter | Underscore | Digit} [?!]
```

String constants

Strings of characters are enclosed in double quotes. Inside of the string, double quotes are denoted as the escape sequence `\"`, backslashes as `\\`. Additionally, some non-printing characters can be written as escape sequences as well:

<code>\n</code>	The newline character (ASCII 10).
<code>\r</code>	The carriage return character (ASCII 13).
<code>\t</code>	The tabulator character (ASCII 8).
<code>\b</code>	The backspace character (ASCII 9).

```
StringConst ::= "{any except Special | \" | \\ | \n | \r | \t | \b}"
Special     ::= " | \"
```

Character constants

Characters are enclosed in single quotes. As in string constants, escape characters can be used in character constants. Only one character (resp. escape sequence) can be in a character constant.

```
CharConst ::= '(any except Special | \" | \\ | \n | \r | \t | \b)'
```

Integer constant

Integer constants are currently limited to the range of long values in the C implementation used to compile Turtle.

```
IntConst ::= Digit{Digit}
```

Real constant

Real constants denote floating point approximations of real numbers. They are currently limited to the range of double values in the C implementation used to compile Turtle.

```
RealConst ::= Digit{Digit}.Digit{Digit}[(e|E)[-|+ ]Digit{Digit}]
```

Operators

The operators are summarized in the Operator Table below.

Reserved Words Table:

module export import

module introduces a module, the others manage the import and export of modules and identifiers.

fun Used for declaring functions, function expressions and function types.

constraint

Used for declaring constraints and constraint expressions.

if then else

Used in if-then-else statements.

while do Used in while-do statements.

end Ends functions, constraints and compound statements.

return Returns values from functions.

var Variable declarations.

type Type declarations.

array list string of

Used in type expressions, and in array, list and string constructors.

and not or

Logical operators.

false true

Truth constants.

require prefer retract

Used for managing the constraint store.

hd tl null

List operators and the empty list constant.

sizeof Generic size operator.

Operator Table:

. Dot. Used to separate the components of qualified identifiers.

,	Comma. Separating items in parameter lists, variable declarations, lists, arrays tuples etc.
()	Parenthesis are used to group elements of value and type expressions, parameter lists etc.
{ }	Curly braces enclose array expressions.
[]	Square brackets are used as indexing operators for using in indexed (array) expressions and enclose list expressions.
!	Annotates constrained types.
;	The semicolon terminates declarations and statements.
+ - * / %	Arithmetic operators.
= < <= > >= <>	Comparison operators.
:=	Assignment operator.
hd tl :	List operators.
sizeof	Generic size operator.

3.1.3 Turtle Syntax

The description of the Turtle grammar is divided in various sections, and each section is provided with some additional comments.

3.1.3.1 Module Syntax

Turtle programs are organized in compilation units. These compilation units are called “Modules”. The head of a module consists of a module header which states the name of the module, and then the (possibly empty) import and export statements, followed by a sequence of declarations. The import statements states which modules will be used in the module, and the export statement lists the bindings which are public. These bindings can be variable, function and constraint definitions as well as type declarations.

```

CompUnit      ::= Module
Module        ::= 'module' QualIdent [ModuleParams] ';'
               ModDecls Declarations
ModuleParams  ::= '<' ModuleParam {',' ModuleParam} '>'
ModuleParam   ::= Ident
ModDecls     ::= Imports Exports
Imports      ::= <empty>
               | 'import' ImportIdent {, ImportIdent} ';'
Exports      ::= <empty>
               | 'export' QualIdent {, QualIdent} ';'
ImportIdent   ::= QualIdent [ImportAnnotation]
ImportAnnotation ::= '<' Type {',' Type} '>'

```

3.1.3.2 Declaration Syntax

Declarations in Turtle can be type, variable, function and constraint declarations. Their syntax is given below. In a program, more than one declarations can have the same name, provided that they can be distinguished by declaration kind (type declarations vs. the other kinds) or by type.

```

Definitions      ::= { Definition ';' }
Definition       ::= TypeDef | DatatypeDef | VarDef | FunDef | ConstraintDef
TypeDef          ::= 'type' Ident = Type
DatatypeDef     ::= 'datatype' Ident '='
                  DatatypeVariant {'or' DatatypeVariant}
DatatypeVariant ::= Ident ['(' VariableList ')']

FieldList       ::= Field {',' Field}
Field           ::= Ident ':' Type

VarDef          ::= 'var' VariableList

VariableList    ::= Variable {',' Variable}
Variable        ::= Ident ':' Type [':=' ConsExpression]

FunDef          ::= 'fun' Ident ParameterList [':' Type] SubrBody
ConstraintDef   ::= 'constraint' Ident ParameterList SubrBody

ParameterList   ::= '(' [Parameter {',' Parameter}] ')'
Parameter       ::= ['out'] Ident ':' Type

Type            ::= QualIdent
                  | '!' Type
                  | '(' ')'
                  | '(' Type {',' Type} ')'
                  | 'array' 'of' Type
                  | 'list' 'of' Type
                  | 'string'
                  | 'fun' '(' [Type {',' Type}] ')' [':' Type]

SubrBody        ::= StmtList 'end'

```

3.1.3.3 Statement Syntax

Statements are the basic entities of which imperative programs are based. Statements in a statement list are executed in top-down direction, and statements can have side effects. In Turtle, there are three statements uncommon in other languages, the constraint statements `require`, `prefer` and `retract`. They are used for maintaining the constraint store.

```

StmtList        ::= { Statement ';' }
Statement       ::= VarDef | FunDef | ConstraintDef
                  | IfStatement | WhileStatement | ReturnStatement
                  | RequireStatement | PreferStatement | RetractStatement
                  | ExpressionStatement
IfStatement     ::= 'if' CompareExpression 'then' {Statement} 'end'
                  | 'if' CompareExpression 'then' {Statement} 'else'
                  {Statement} 'end'
WhileStatement  ::= 'while' CompareExpression 'do' {Statement} 'end'
ReturnStatement ::= 'return' [TupleExpression]
RequireStatement ::= 'require' CompareExpression
PreferStatement ::= 'prefer' CompareExpression
RetractStatement ::= 'retract' CompareExpression
ExpressionStatement ::= Expression

```

3.1.3.4 Expression Syntax

Expressions consist of assignment expressions, comparing and boolean expressions and arithmetic expressions. Additionally, functions and constraints can result from evaluating expressions, such as in functional programming languages.

```

Expression      ::= AssignmentExpression
AssignmentExpression ::= TupleExpression [':' TupleExpression]

TupleExpression ::= ConsExpression {' ',' ConsExpression}

ConsExpression  ::= CompareExpression [':' ConsExpression]

CompareExpression ::= AddExpression [CompareOp AddExpression]
AddExpression     ::= MulExpression {AddOp MulExpression}
MulExpression     ::= Factor {MulOp Factor}
Factor            ::= SimpleExpression
                  | '-' Factor
                  | 'not' Factor
                  | 'hd' Factor
                  | 'tl' Factor
                  | 'sizeof' Factor

SimpleExpression ::= AtomicExpression {ActualParameters | Index}
ActualParameters ::= '(' [ConsExpression {' ',' ConsExpression}] ')'
Index             ::= '[' AddExpression ']'

AtomicExpression ::= QualIdent
                  | IntConst | RealConst | StringConst | CharConst
                  | BoolConst | ArrayExpr | ListExpr
                  | FunExpression | ConstraintExpression
                  | 'array' AddExpression 'of' TupleExpression
                  | 'list' AddExpression 'of' TupleExpression
                  | 'string' AddExpression 'of' SimpleExpression
                  | '(' TupleExpression ')'
                  | 'null'

BoolConst       ::= 'false' | 'true'

FunExpression   ::= 'fun' ParameterList [':' TypeExpr] SubrBody
ConstraintExpression ::= 'constraint' ParameterList SubrBody
ArrayExpr       ::= '{' [{ConsExpression}
                       {' ',' ConsExpression}] '}'
ListExpr        ::= '[' [{ConsExpression}
                       {' ',' ConsExpression}] ']'

MulOp           ::= '*' | '/' | 'and'
AddOp           ::= '+' | '-' | 'or'
CompareOp       ::= '=' | '<>' | '<' | '<=' | '>' | '>='

```

3.1.3.5 Basic syntax items

In the beforementioned syntax descriptions, the following non-terminals have been used:

```

QualIdent      ::= Ident {'.' Ident}

```

3.2 Turtle semantics

This section documents informally the static and dynamic semantics of Turtle programs.

3.2.1 Expression Types

Arithmetic operators (AddOp and MulOp in the grammar, and unary `-`) may only be applied to operands of numeric type, and the type for both operands of the binary operators must be the same. The result type is the same type as the types of the operands.

The comparison operators may only be applied to operands of comparable types, which are integers, reals, characters and booleans. Both operands must be of the same type. They yield a result of type boolean.

The operators `and`, `or` and `not` may only be applied to operands of boolean types. The result type is boolean, too. The evaluation of `and` and `or` expressions is short-circuited, that means that if in an `and` expression the first operand evaluates to `false`, the second operand is not evaluated, because the result of the overall expression is known. Similarly, if the first operand to `or` evaluates to `true`, there is no need to evaluate the second operand.

3.2.2 List operators

The operators `hd` and `tl` return the head or tail of a list value, respectively. If the list operand is `null` (the empty list), an exception is raised.

The `:` (pronounced “cons”) operator takes an expression and a list expression and creates a new list cell with the expression as the list head and the list expression as the list tail.

In addition to the cons operation, lists can be created as documented in Section 3.2.4 [List and array expressions], page 15.

The empty list can be detected by comparing a list expression against `null`: all lists are terminated by this constant.

3.2.3 Array operators

Arrays are aggregations of homogenous objects. Arrays can have varying lengths, but the length must be specified when creating the array value. Arrays can be constructed by the means documented in Section 3.2.4 [List and array expressions], page 15.

Array elements are addressed by their index, which must be in the range $0 \dots N-1$, where N is the length of the array. An array index expression looks like this:

`A[2]`

This expressions references the array element with index 2 (the *third* element, because of zero-based indexing). If written on the lefthand side of an assignment operator (`:=`), an array element can be overwritten with the expression on the right hand side.

This indexing can be used for string in the same way.

The length of an array expression (or string) can be determined with the `sizeof` operator.

3.2.4 List and array expressions

All elements of list and array expressions must have the same type. The type of a list or array expression is `list of element type` or `array of element type`, respectively. The empty list expression `[]` is equivalent to the constant `null` and is compatible with all list types. The empty array expression `{}` is compatible with all array types.

The special constant `null` is compatible to all list types.

The array constructor `array expr of expr` has the type `array of element type`, where *element type* is the type of the second expression of the constructor. The first expression must be of type `int`. The constructor returns an array of the length given as the first expression, where each list element is initialized to the second expression.

The list constructor `list expr of expr` has the type `list of element type`, where *element type* is the type of the second expression of the constructor. The first expression must be of type `int`. The constructor returns a list of the length given as the first expression, where each list element is initialized to the second expression.

The string constructor `string expr [of expr]` has the type `string`. The first expression must be of type `int`, the second expression, if given, must be of type `char`. If omitted, the contents of the newly created string value is unspecified.

3.2.5 Return statements

Return statements are not followed by an expression if they are contained in a function of return type `void`. Otherwise, the expression must have a type compatible with the return type of the enclosing function. A function with non-void return type *must* have a return expression in tail position.

3.2.6 Overloading

A Turtle module may declare more than one type, variable, function or constraint with the same name, as long as they are distinguishable by identifier kind (type vs. variable, function or constraint), or by type. Consider the following Turtle fragment:

```
var a: int, a: real;
a := 1;
a := 1.1;
```

This code is legal, because the compiler can determine unambiguously which variable is used in the second and third line. The second line refers to the `int` variable, because an `int` value is assigned, and the third line modifies the `real` variable.

Turtle's overloading is more powerful than in C++ or Java, because not only the parameter types of functions can be overloaded but also the return type and variables which don't have function types. Because of these restrictions in C++/Java, the above code would not work there. Turtle is similar to Ada in that respect.

When type-checking programs, the Turtle compiler considers every possible interpretation of a given expression, and then tries to find a unique type-consistent interpretation. If this is not possible, an error message is issued, either stating that no possible typing was found or that the expression is ambiguous, that means, there is more than one legal interpretation. Consider this example:

```
var a: int, a: real;
var b: array of int, b: array of real;
b := {a, a};
```

Here, both the assignment of a two-element array of integers and reals to an integer or real array variable is legal. The compiler will signal an error for this example.

3.2.7 Generic Modules

Turtle supports so-called generic modules. They provide a restricted form of polymorphism. Modules can be declared to have module parameters, which are types. In the module, these parameters can be used for defining functions and data types. When a generic module is then imported, it must be instantiated, that is, actual types must be provided for the module parameters.

The exported functions of the module are then available with the actual type substituted for the module parameter.

A module can be imported more than once into the same scope, provided that the combination of module parameters differ.

Module parameters can be used to form the actual types in module import statements, too.

3.3 Data Types

Turtle provides a variety of builtin data types, such as integers, reals, strings etc., and support for arrays and lists is also built into the language. Additionally, Turtle provides a powerful mechanism for defining new data types for specific purposes. This functionality is used in the standard library, which provides a set of abstract data types such as binary trees, hash tables and so on.

This subsection documents the basic data types, compound data types and user-defined data types.

3.3.1 Basic data types

Basic data types provide the most basic functionality for computing. For the numeric data types (integers and reals), arithmetic operations are defined, strings can be decomposed to characters and so on.

3.3.1.1 Integers

Integer numbers are numbers without a fraction part.

Integer numbers are written as a non-empty sequence of the digits 0 to 9. The name of the data type is `int`, which is not a reserved word but defined in the standard outermost compilation environment.

```
1
10
1000
424242
23
```

Turtle defines the arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (modulo). Integers can be compared with the operators `=`, `<>`, `<`, `>`, `<=` and `>=`.

The module `ints` (see Section 5.3.1 [ints module], page 33) provides some useful constants and functions for integer values.

3.3.1.2 Longs

While there is an integer data type called `int`, there is an additional integer data type supported, called `long`. The difference between `int` and `long` is that integer values are restricted to the range `ints.min..ints.max`, which is not necessarily the complete 32-bit value space. `long` values, on the other side, are guaranteed to have at least 32 bits, on 64 bit platforms, it can be even 64 bits.

Long integer constants are written like integer constants, but must be appended with an upper-case L, as in the following examples:

```
42L
23L
0L
2147483647L
```

Note especially the last example, which is too large to fit into an `int` value on 32 bit systems.

The same arithmetic operations as supported for `int` values are supported, that is, `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (modulo). Long integers can be compared with the operators `=`, `<>`, `<`, `>`, `<=` and `>=`.

The module `longs` (see Section 5.3.2 [longs module], page 34) provides some useful constants and functions for integer values.

3.3.1.3 Reals

In contrast to integers, reals can have a fraction part. The fraction part is separated from the leading sequence of digits by a period, and can be followed by an optional exponent part, like in C or Java. The name of the data type is `real`, which is not a reserved word but defined in the standard outermost compilation environment.

```
1.0
3.14159
1.0e12
20E-12
```

Turtle defines the arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). Reals can be compared with the operators `=`, `<>`, `<`, `>`, `<=` and `>=`.

The module `reals` (see Section 5.3.3 [reals module], page 35) provides some useful constants and functions for real values.

3.3.1.4 Booleans

The booleans represent the two truth values *true* and *false*. They are denoted by the reserved words `true` and `false`. The name of the data type is `bool`, which is not a reserved word but defined in the standard outermost compilation environment.

Boolean values can be compared with `=` and `<>`. The operations `and` for conjunction, `or` for disjunction and `not` for negation can be used with expression of type `bool`. The `and` and `or` are evaluated with short-circuiting, that is if the first operand of `and` evaluates to false, the second operand is not evaluated, and if the first operand of `or` is true, the evaluation of the second is skipped.

3.3.1.5 Characters

Characters are used for storing character data. Character constants are denoted by enclosing a character in single quotes, as in C/C++/Java. For denoting some special characters, the following escape sequences are defined:

```
'\"'      Denotes a double quote.
'\''      Denotes a single quote.
'\'\'     Denotes a backslash character.
'\n'      Denotes a newline character.
'\r'      Denotes a carriage return character.
'\t'      Denotes a tab character.
'\b'      Denotes a backspace character.
```

The name of the data type is `char`, which is not a reserved word but defined in the standard outermost compilation environment.

The module `chars` (see Section 5.3.5 [chars module], page 36) provides some useful constants and functions for character values.

3.3.1.6 Strings

Strings are sequences of characters. They have a fixed length, which is specified when creating a new string value, or when writing down a string literal. String literals consist of zero or more characters, enclosed in double quotes. They may contain the same escape sequences as character literals Section 3.3.1.5 [Characters], page 18, but they may not span a line boundary, that is they may not contain a character with the character code of `'\n'`.

Strings are different than the former documented basic data types, because they contain zero or more values of another data type, character. The reason for building this data type into Turtle is that it is so common, that using (for example) `array of char` or `list of char` is considered too inconvenient.

The name of the data type is `string`, which is, unlike the other type names so far, a reserved word.

Elements of a string can be extracted by applying the subscript operator, `[]`, to a string expression:

```
var s: string := "Hallo";
io.put (s[1]);    // Writes 'a' to stdout.
```

The same operator, when applied to the lefthand side of an assignment, will change an element of the string.

```
var s: string := "Hallo";
io.put (s[1]);    // Writes 'a' to stdout.
s[1] := 'u';
io.put (s);       // Writes "Hullo" to stdout.
```

Strings are either created by writing down string literals, like this:

```
"           // Empty string.
"Hello"
"a"
```

or with a string constructor expression, which creates a string of a given length and initializes all elements to a given character:

```
var s: string := string 12 of '!';
```

The length of a string can be determined with the `sizeof` operator.

The module `strings` (see Section 5.3.6 [strings module], page 37) provides some useful constants and functions for string values.

3.3.2 Compound data types

Turtle provides three data types which are made up of other data types: arrays, lists and function types.

3.3.2.1 Array data type

An array data type is written as `array of basetype`.

The length of an array is fixed when the array value is created and can be determined later with the `sizeof` operator.

Elements of an array can be fetched from the array value or stored into it using the subscript operator `[]`.

Arrays can be created with array constructor expressions or by writing down array expressions.

Array constructor expressions create arrays of a specified length, where each element of the array is initialized to the same value. It is important to note that the array elements are really

initialized to the *same* value, so that if this value is of an array, list, string or user-defined data type, the same value is shared by all array elements and by modifying one element, all other array elements are affected as well.

```
var a: array of int := array 20 of 23;
var l: array of string := array 12 of "Hello";
```

Array expressions are used to create array values with pre-defined contents. They are written as comma-separated lists of array elements, enclosed in curly braces.

```
var a: array of int := {1, 3, 12, 4};
var a: array of string := {"0", "sole", "mio"};
```

The library module `arrays` (see Section 5.5.1 [arrays module], page 43) provides some useful and functions for array values. The library also contains the module `arraysort` (see Section 5.5.4 [arraysort module], page 44) for sorting arrays of arbitrary contents and the module `arraysearch` (see Section 5.5.3 [arraysearch module], page 44) for searching arrays linearly or by bisection. The library module `arraymap` (see Section 5.5.2 [arraymap module], page 44) exports a function for applying functions to all elements of an array in turn, returning an array of the function results

3.3.2.2 List data type

A list data type is written as `list of basetype`.

The length of a list is variable, a list can be extended by prepending values to the front of the list.

The first element of a list (the head) is fetched from a list with the prefix operator `hd`, the list of the remaining elements (the tail) is fetched with the operator `tl`. An element is added to the front with the so-called *cons* operator, written as `::`.

Lists can be created with list constructor expressions or by writing down list expressions.

List constructor expressions create lists of a specified length, where each element of the list is initialized to the same value. It is important to note that the list elements are really initialized to the *same* value, so that if this value is of an array, list, string or user-defined data type, the same value is shared by all list elements and by modifying one element, all other list elements are affected as well.

```
var l: list of int := list 20 of 23;
var l: list of string := list 12 of "Hello";
```

List expressions are used to create lists with pre-defined contents. They are written as comma-separated lists of list elements, enclosed in square brackets.

```
var l: list of int := [1, 3, 12, 4];
var l: list of string := ["0", "sole", "mio"];
```

The library module `lists` (see Section 5.4.1 [lists module], page 40) provides some useful functions for lists. The library also contains the module `listsort` (see Section 5.4.8 [listsort module], page 43) for sorting lists of arbitrary contents and the module `listsearch` (see Section 5.4.7 [listsearch module], page 43) for searching lists. The library module `listmap` (see Section 5.4.2 [listmap module], page 41) exports a function for applying a function to all elements of a list, producing a list of the function's results.

3.3.2.3 Function data type

A function data type is written as `fun(paramtypes...): returntype`

Functions can be created with function expressions or by function declarations..

The standard module `compose` (see Section 5.1.3 [compose module], page 26) provides a function for composing functions.

3.3.2.4 Tuple data type

Values can be composed to form tuples. Tuples are ordered sequences of fixed length, where the data types of the elements can be different.

A tuple type is written by enclosing more than one type expression in parentheses, separated by commas, for example:

```
(int, string, array of string)
```

Tuple values are created by writing down comma-separated list of more than one expression.

Tuple values are decomposed by assigning to a tuple of variables, such as in:

```
var a: int, b: int, c: (int, int);
  c := 1, 2;
  a, b := c;
```

For programming convenience, the standard library contains two modules which export functions for dealing with two very common tuple types. The modules `pairs` (see Section 5.6.1 [pairs module], page 45) and `triples` (see Section 5.6.2 [triples module], page 45) export selector functions for tuple components.

3.3.3 User-defined data types

User-defined data types are defined using the `datatype` declaration. As an example, we will define a data type representing a binary tree. The name of the new type will be `tree`, and there are two variants for objects of this type: First, there will be a variant for interior node, which consist of a left and a right subtree, and second there must be a variant for the leafs of the tree. This is how such a datatype decleration looks like in Turtle:

```
datatype tree = node (left: tree, right: tree) or
                leaf (value: int);
```

There are several things to note here. The variants of the data type are listed on the right hand side of the declaration, separated by the reserved word `or`. User-defined data types can be recursive, that means that the left hand side of the declaration may appear on the right hand side as well.

Each variant has a number of data fields. This field list can be empty, and if it is empty, the parenthesis can be omitted. Fields in different variants can have the same name and must not have the same types if they have the same name, but the fields of one variant must be distinct either in name or in type.

A datatype declaration causes the compiler to create several functions automatically:

Constructor functions

For each variant a constructor function is created which can be called to create objects of this variant. Constructor functions are called like the variant and expect the fields' data types as arguments.

Discriminator functions

For each variant a discriminator function is created which expects an object of the data type and returns `true` if the object is of the correct variant, and `false` otherwise. The name of the discriminator function is the name of the variant with an appended `?` (question mark).

Selector functions

For each name/type combination of the fields a selector function is created which extract the field from an object of the data type. If the object is of a variant which does not have a field with the name/type combination, an exception is raised. The name of the selector functions is simply the name of the fields.

Setter functions

For each name/type combination of the fields a selector function is created which sets the field to an object of type of the field. If the object is of a variant which does not have a field with the name/type combination, an exception is raised. The name of the setter functions is derived from the field names by appending a ! (exclamation point).

For our example above, the following functions are automatically created:

```
// Constructors
fun node (tree, tree): tree;
fun leaf (int): tree;

// Discriminators
fun node? (tree): bool
fun leaf? (tree): bool

// Selectors
fun left (tree): tree;
fun right (tree): tree;
fun value (tree): int

// Setters
fun left! (tree, tree);
fun right! (tree, tree);
fun value! (tree, int);
```

3.4 Runtime environment

The Turtle runtime environment consists of a standard virtual machine like the ones used for other imperative programming languages, enriched with a constraint solver interface, a constraint store and several constraint solvers for solving primitive constraints. User-defined constraints are either translated by the compiler or dynamically by the runtime library to primitive constraints, which are then solved using the primitive solvers.

4 Constraint Programming

Turtle is a constraint-imperative language. That means that the normal imperative language constructs found in today's imperative programming languages are present in Turtle as well as more declarative language elements. This chapter gives an introduction to constraint programming in Turtle, will describe the principles of constraint programming and the interaction between imperative and constraint features, and will give some hints about what can be accomplished with constraints in Turtle.

4.1 Constraints and the Constraint Store

Before we jump into constraint programming, we have to give a brief introduction to constraints and the mechanisms necessary to manage and solve constraint systems.

One basic notion is the *constraint*. A constraint is a formula, in Turtle it is defined to be either an expression of type boolean, or the application of a user-defined constraint. All constraints which are to be solved in a program are added to the so-called *constraint store*, which is simply a set of constraints.

In Turtle, constraints are added to or removed from the store by using *constraint statements* (see Section 4.3 [Constraint Statements], page 23). Whenever a constraint is added to or removed from the store, the constraint solver tries to enforce as many constraints as possible.

Since Turtle supports *constraint hierarchies*, that is, constraints of different strength, each constraint in the constraint store is labelled with its associated strength. Strengths are given to constraints when adding them to the store. One special strength is called *required*, and constraints of this strength must hold during program execution, or otherwise an exception will be raised.

4.2 Constraintable Variables

In Turtle, variables are either normal variables or *constraintable variables*. Constraintable variables have the special property that they can not be set by assignment statements, but only by constraints. The following example will demonstrate the difference:

```
var x: int;
x := 2;
```

'x' is a normal variable, and is set to the value 2 by assignment. In the next example, the variable 'y' is declared as a constraintable variable (note the ! (exclamation point) in the type declaration of the variable), and can thus be used in a constraint to set its value.

```
var x: !int;
require y = 2;
```

4.3 Constraint Statements

The language Turtle knows about three statement kinds normally not found in imperative languages: **require**, **prefer** and **retract** statements. These commands are used to maintain the constraint store (see Section 4.1 [Constraints and the Constraint Store], page 23).

This example adds a constraint to the store:

```
require x > 3;
```

Executing this constraint statements will cause the Turtle runtime to add the constraint 'x > 3' to the store, and to try to satisfy it. If the current value of the variable 'x' is greater than 3, nothing will happen. If not, an exception will be raised saying that a required constraint was not satisfied. The statement **prefer**, on the other hand, will only try to satisfy a constraint if possible. For the constraint example above, this would not make sense, but if instead a constraintable variable

was involve in the constraint, the solver would try to satisfy it as well as possible by adjusting the variable's value:

```
var x: !int := 2;
prefer x > 3;
```

The constraint will set the variable to 4, because this will cause the constraint to be satisfied.

4.4 Constraints as Assertions

For the start, constraints can simply be used to code checks for program invariants. If a utility function you write expects that an integer parameter be always greater than zero, you can add the constraint statement

```
require x > 0;
```

at the beginning of the function. This is similar to the use of the `assert()` macro in C source code. Since no constrainable variables appear in the constraint, the constraint does not get added to the constraint store, but only tested. If the test fails, an exception is raised. In this way, preconditions can be tested by adding constraint statements at the beginning of functions, and postconditions by adding such statements at the end, just before any `return` statements.

Of course this is not real constraint programming, but it shows that sometimes new features can be adapted to old needs.

5 Standard library

The Turtle standard library comes with a couple of modules for commonly needed programming tasks, such as input/output, string handling, list manipulation etc.

The functions in the standard library are documented like shown below:

bla (*foo*: **bar**): **baz** Library function
 Description of function. . .

This means that a library function called **bla** is documented. The function expects one parameter, called *foo* which is of type **bar**. The type of the return value is called *baz*. The function header is then followed by a description of what the function does, and which preconditions must be fulfilled when calling the function.

Remember that in Turtle all references to functions and variables in other modules must be fully qualified, so if you want to call the function **map** of module **listmap** from your main program, you have to write:

```
l := listmap.map (proc, l);
```

5.1 General modules

This section documents general useful modules, which are not closely related to specific data structures or application fields like input/output or systems programming.

5.1.1 math module

This module provides mathematical constants and functions.

pi : **real** Constant
 This is the constant **pi**.

sin (*x*: **real**): **real** Function
asin (*x*: **real**): **real** Function
cos (*x*: **real**): **real** Function
acos (*x*: **real**): **real** Function
tan (*x*: **real**): **real** Function
atan (*x*: **real**): **real** Function
atan (*x*: **real**, *y*: **real**): **real** Function

These are the common trigonometric functions. They are mapped directly to the functions in the C library.

5.1.2 random module

The exported functions **rand** delivers a random number in the range **0..ints.max**. The current implementation uses the C library function **rand()** for obtaining the random numbers and the C library function **srand()** for seeding the generator.

seed (*seed*: **int**) Function
 Seed the random number generator with the given seed value *seed*.

rand (): **int** Function
 Return a random number in the range **0..ints.max**.

5.1.3 compose module

This module exports one function, `compose`, which implements function composition.

compose (*f*: fun(inter): res, *g*: fun(arg): inter): fun(arg): res Function
 Return a function of one argument, which first applies *g* to its argument and then *f* to the result of this function application.

5.1.4 identity module

This module exports the function `id`, which is the identity on values of the type given as the module parameter *A*.

id (*data*: A): A Function
 Return the argument *data* unchanged. This function is especially useful for higher-order functions which expect a function to apply to all members of a collection, such as `map`. For example,

```
l := listmap.map (identity.id, l)
```

can be used to copy a list.

5.1.5 compare module

The module `compare` exports several comparison functions, where each of the functions compares two values of a basic data type. These functions are especially useful as parameters for the higher-order functions, such as the sorting functions in the `listsort` or `arraysort` modules.

cmp (*x*: bool, *y*: bool): int Function
cmp (*x*: int, *y*: int): int Function
cmp (*x*: long, *y*: long): int Function
cmp (*x*: real, *y*: real): int Function
cmp (*x*: char, *y*: char): int Function
cmp (*x*: string, *y*: string): int Function
 Compare the values *x* and *y*, and return -1 if the first argument is to be considered smaller, 0 if they are equal, and 1 if the first argument is greater than the second.

5.1.6 option module

This module exports one data type, `option`, and the corresponding constructor, accessor and discriminator functions.

The `option` type is intended to be the result of partial functions, which can either succeed and return a useful value, or fail. The former result will be of variant `some` with the useful packaged in the `data` field, whereas the latter will yield a value of variant `none`.

This is (of course) inspired by the `option` type in ML.

option Data type
 Defined as:

```
datatype option<A> =
  none or
  some(data: A)
```

The `option` data type. It either represents nothing (variant `none`), or a value of the data type *A*, which is a parameter to this module.

5.1.7 cmdline module

The function `getopt`, exported by this module, deconstructs a command line into options, option parameters and other parameters.

optspec Data type

Defined as:

```
datatype optspec =
  flag(flag_char: char, flag_string: string) or
  option(option_char: char, option_string: string)
```

Values of type `optspec` define the possibilities of command line flags (parameterless options) and options (which have parameters).

option Data type

Defined as:

```
datatype option =
  flag(flag_char: char) or
  option(option_char: char, argument: string) or
  parameter(param: string)
```

Calls to the `getopt` function return lists of this type. Each element of the list stands for one of three different argument types:

Flag A flag is a parameterless option, which acts as a switch.

Option An option has an associated argument, which gives the program additional information about the option.

Parameter Everything not beginning with one or two dashes is considered a normal parameter, such as a filename to act on. Everything following the special option `--` will also be treated as a parameter, even if it starts with a dash. Note that a single dash is also a normal parameter.

getopt (*specs*: list of `optspec`, *args*: list of `string`): list of `option` Function

Given a list of option specifications and a list of command line arguments (as passed to the `main` function, for example), return a list of options, classified and deconstructed as flags, options or parameters.

5.1.8 hash module

This module exports hash functions for the basic data types.

hash (*i*: `int`): `int` Function
hash (*l*: `long`): `int` Function
hash (*r*: `real`): `int` Function
hash (*b*: `bool`): `int` Function
hash (*c*: `char`): `int` Function
hash (*s*: `string`): `int` Function

Calculate a hash value for the given argument. The returned value is in the range `0 .. ints.max`.

5.1.9 hashtab module

This is a generic implementation for hash table. The range and domain types for the partial mapping implemented by the hash table are supplied as module parameters and the hash function for mapping the keys to integers must be given when constructing a hash table.

The module expects two module parameters, *range* and *domain*, where *range* is the type of the keys in the hash table and *domain* is the type of the associated values.

hashtable Data type

Defined as:

```
datatype hashtable =
  tab(size: int, data: array of bucket, hashfn: fun(range): int, cmpfn: fun(ran
```

This is the hashtable data type. It stores the hashing and comparison functions to be used with the keys, so that these functions must not be passed to the insertion/search functions each time they are invoked.

make (*hashfn*: fun(*range*): int, *cmpfn*: fun(*range*, *range*): bool): Function
hashtable

Create a new hash table which maps keys of type *range* to values of type *domain*. *hashfn* is a function which determines the hash value of a key and *cmdfn* is a function which determines whether two keys are actually the same.

insert (*tab*: hashtable, *key*: range, *val*: domain) Function
Insert the *key/value* pair into the hash table *tab*. If *key* is already in the table, its value is overwritten.

delete (*tab*: hashtable, *key*: range) Function
Remove the entry with key *key* from the hash table *tab*. Do nothing, if the key is not present in the table.

lookup (*tab*: hashtable, *key*: range): option.option Function
Search the hash table *tab* for an entry with key *key* and return `some(value)` if the key was found, and `none()` if not found in the table.

5.1.10 trees module

Functions for creating and inspecting binary trees of type *A*, where *A* must be instantiated when importing this module.

tree Data type

Defined as:

```
datatype tree<A> =
  nil or
  node(data: A, left: tree, right: tree)
```

Data type for binary trees. The empty tree is created with a call to the constructor `nil`, and a non-empty tree with the constructor `node`.

node (*d*: A): tree Function
Utility function for creating a singleton tree.

preorder (*t*: tree, *f*: fun(A)) Function

inorder (*t*: tree, *f*: fun(A)) Function

postorder (*t*: tree, *f*: fun(A)) Function

Iterate over the tree *t* in preorder, inorder or postorder, respectively and call function *f* at every node, with the data element of the node as argument.

copy (*t*: tree): tree Function
 Create a deep copy of the tree *t*.

5.1.11 bstrees module

Functions for creating and inspecting binary search trees with a key of type *A*, and a data object of type *B*, where *A* and *B* must be instantiated when importing this module.

tree Data type
 Defined as:

```
datatype tree<A, B> =
  tree(cmpfn: fun(A, A): int, root: trees.tree<(A, B)>)
```

Data type for binary search trees.

tree (*cmpfn*: fun(A, A): int): bstrees.tree<A, B> Function

insert (*t*: bstrees.tree<A, B>, *key*: A, *val*: B): bstrees.tree<A, B> Function
 Insert the pair (*key*, *val*) into the binary search tree *t* and return the updated tree. The old copy of the search tree remains usable; the binary search tree implemented by this module is persistent.

search (*t*: bstrees.tree<A, B>, *key*: A): option.option Function
 Search the binary search tree *t* for the key *key* and return the associated value, if found, packaged in an option.option type. If not found, the variant option.none is returned.

5.1.12 bintree module

Simple, but working binary tree implementation for building search trees. A binary tree is parametrized by two types, one for the key and one for the associated value. Both the insertion and the search function require a comparison function to be passed, so that an order on the keys can be established.

The module expects two module parameters, *A* and *B*, where *A* is the type of the keys in the search tree and *B* is the type of the associated values.

tree Data type
 Defined as:

```
datatype tree<A, B> =
  empty or
  leaf(element: A, data: B) or
  node(left: tree, right: tree, key: A)
```

The binary search trees are made up of this data type.

insert (*t*: bintree.tree<A, B>, *key*: A, *data*: B, *cmp*: fun(A, A): int): bintree.tree<A, B> Function

Return a new search tree in which all key/value pairs from the input tree *t* are stored and additionally a pair of the parameters *key* and *data* is stored. *cmp* is a comparison function used to determine the order in the tree. *cmp* is expected to return a value less than 0 if the first argument is smaller than the second, a value greater than 0 if the first argument is greater and 0 if they are equal.

find (*t*: bintree.tree<A, B>, *key*: A, *cmp*: fun(A, A): int): option.option Function

Search the binary search tree *t* for an entry with key *key*. Return option.some (data) with the data value associated with *key* if *key* was found, otherwise return option.none (). Note that it is not specified which data value will be returned if the key *key* appears more than once in the tree.

5.1.13 binary module

Support module for binary (byte-) arrays.

binary Data type

Defined as:

```
type binary = internal.binary.binary
```

The `binary` data type is an alias for the type of the same name from the module `internal.binary`, where the real type and function definitions reside.

make_binary : fun(int): binary Constant

binary_get : fun(binary, int): int Constant

binary_set : fun(binary, int, int) Constant

binary_size : fun(binary): int Constant

These functions create binary arrays of a given size, extract an element from these arrays or stores an integer into a specified location. `binary_size` returns the number of elements in the binary array `b`.

to_string (*b*: internal.binary.binary): string Function

Convert the binary array `b` to a string by simply converting the integer values in `b` to characters using the function `chars.chr`.

from_string (*s*: string): internal.binary.binary Function

Convert the string `s` to a binary array by converting the characters in the string to their code values using the function `chars.ord`.

5.1.14 exceptions module

This module exports some functions for raising and handling exceptions.

Exceptions are raised by calling the function `exceptions.raise`, which has the same effect as performing some illegal operation such as taking the head of the empty list. The argument to `raise` is the name of the exception.

Exception handling is done by calling the function `exceptions.handle`. The first argument is a functions which might possibly raise an exception, while the second is a function which will be called when an exception occurs. If no exception is raised, `handle` returns without calling the handler function.

raise (*s*: string) Function

Raise an exception with name `s`.

handle (*thunk*: fun(): (), *handler*: fun(string)) Function

Call the function `thunk`. If any exception is raised while `thunk` is running, the function `handler` will be called with the exception name as the only argument. When `handler` returns, it will return to the caller of `exceptions.handle`, in the same way as the call would return when `thunk` was returning without an exception.

null_pointer_ex (): string Function

out_of_range_ex (): string Function

subscript_ex (): string Function

wrong_variant_ex (): string Function

require_ex (): string Function

The return value of these functions is the corresponding exception name, which is the same that would be used if the illegal operation would be performed.

That means that

```
exceptions.raise (exceptions.subscript_ex ())
```

has the same effect as

```
var s: string;
s[-1] := 'a'
```

5.1.15 filenames module

Library module for filename manipulation functions.

path_seperator : char	Variable
The path name element separator used by the module.	
basename (s: string): string	Function
basename (s: string, ext: string): string	Function
Return the filename <i>s</i> without any directory component. If the parameter <i>ext</i> is specified and matches the file name extension of <i>s</i> , this is removed also.	
dirname (s: string): string	Function
Return the directory component of the filename <i>s</i> , without any trailing path separator.	

5.2 Input and output modules

This section collects all input/output related modules of the standard library. The most basic module in this group is `io`, which provides basic input and output operations for basic data types.

5.2.1 io module

This module provides basic input and output functions for some of the builtin data types. Also some functions for handling files are defined.

file	Data type
Defined as:	
<pre>datatype file = file(fd: int, buf: string, pos: int, fill: int, writable: bool)</pre>	
The <code>file</code> data type represents input and output streams. Values of this type are either found in one of the pre-defined variables <code>input</code> , <code>output</code> or <code>error</code> , or are obtained by calling functions like <code>open</code> .	
input : file	Variable
Standard input file.	
output : file	Variable
Standard output file.	
error : file	Variable
Standard output file for error messages.	

put (<i>f</i> : file, <i>s</i> : string)	Function
put (<i>s</i> : string)	Function
put (<i>f</i> : file, <i>i</i> : int)	Function
put (<i>i</i> : int)	Function
put (<i>f</i> : file, <i>l</i> : long)	Function
put (<i>l</i> : long)	Function
put (<i>f</i> : file, <i>r</i> : real)	Function
put (<i>r</i> : real)	Function
put (<i>f</i> : file, <i>b</i> : bool)	Function
put (<i>b</i> : bool)	Function
put (<i>f</i> : file, <i>c</i> : char)	Function
put (<i>c</i> : char)	Function
Write the string representation of the argument to the file <i>f</i> or the standard output file, respectively.	
put (<i>f</i> : file, <i>ls</i> : list of string)	Function
put (<i>ls</i> : list of string)	Function
Write all strings in the argument list to the given file, or standard output, respectively. The strings are separated by newline characters in the output.	
nl (<i>f</i> : file)	Function
nl ()	Function
Terminate the current output line by writing a newline character to the given file or standard output, respectively.	
putln (<i>s</i> : string)	Function
putln (<i>f</i> : file, <i>s</i> : string)	Function
putln (<i>c</i> : char)	Function
putln (<i>f</i> : file, <i>c</i> : char)	Function
putln (<i>i</i> : int)	Function
putln (<i>f</i> : file, <i>i</i> : int)	Function
putln (<i>l</i> : long)	Function
putln (<i>f</i> : file, <i>l</i> : long)	Function
putln (<i>r</i> : real)	Function
putln (<i>f</i> : file, <i>r</i> : real)	Function
putln (<i>b</i> : bool)	Function
putln (<i>f</i> : file, <i>b</i> : bool)	Function
Write the textual representation of the argument to standard output or the given file, respectively. Then terminate the output with a newline character.	
get (<i>f</i> : file): string	Function
get (): string	Function
get (<i>f</i> : file): int	Function
get (): int	Function
get (<i>f</i> : file): bool	Function
get (): bool	Function
get (<i>f</i> : file): char	Function
get (): char	Function
get (<i>f</i> : file): real	Function
get (): real	Function
Read the string representation of a value matching the return type, convert it, and return that value. If the string cannot be converted, return an unspecified value.	
The string reading functions returns <code>null</code> when the end-of-file is reached.	

get (<i>f</i> : file): list of string	Function
get (): list of string	Function
Read all lines from the file <i>f</i> and return them as a list of strings. For an empty file, return null.	
unset (<i>f</i> : file, <i>c</i> : char)	Function
unset (<i>c</i> : char)	Function
Put the character <i>c</i> back into the input file <i>f</i> or standard input, respectively.	
open (<i>name</i> : string): file	Function
Open the file called <i>name</i> for reading. Return a <code>file</code> object for reading from the opened file, or return null if the file can't be opened.	
create (<i>name</i> : string): file	Function
Create a new file called <i>name</i> and open it for writing. Return a <code>file</code> object for writing to the opened file, or return null if the file can't be opened.	
close (<i>f</i> : file)	Function
Close the file object <i>f</i> . After calling this function, <i>f</i> may not be used for file operations anymore.	
flush (<i>f</i> : file)	Function
Flush all buffered output to the underlying operating system file descriptor.	

5.3 Data type related modules

This section documents the modules for handling values of the various basic data types. The modules for the numeric data types export some important constants and utility functions, for example for determining the minimum or maximum of two values. The character handling module exports conversion functions, and the string module functions for deconstructing, composing and examining strings, etc.

5.3.1 ints module

This is a library module exporting integer data type related constants and functions.

min : int	Constant
<code>min</code> is the smallest representable integer value. This value may (and most probably will) differ from the minimum integer value representable on the underlying hardware.	
max : int	Constant
<code>max</code> is the largest representable integer value. This value may (and most probably will) differ from the maximum integer value representable on the underlying hardware.	
min (<i>x</i> : int, <i>y</i> : int): int	Function
Return the minimum of two integer values.	
max (<i>x</i> : int, <i>y</i> : int): int	Function
Return the maximum of two integer values.	
from_string (<i>s</i> : string): int	Function
Convert the string value <i>s</i> to the integer number it represents. If <i>s</i> does not represent any integer number, or if the resulting integer is not in the range <code>ints.min...inits.max</code> , the return value is unspecified.	
to_string (<i>i</i> : int): string	Function
Convert the integer value <i>i</i> to its string representation.	

to_real (<i>i</i> : int): real	Function
Convert the integer value <i>i</i> to a real value.	
from_real (<i>r</i> : real): int	Function
Convert the real value <i>r</i> to an integer value. Decimal places are stripped.	
even? (<i>i</i> : int): bool	Function
odd? (<i>i</i> : int): bool	Function
zero? (<i>i</i> : int): bool	Function
positive? (<i>i</i> : int): bool	Function
negative? (<i>i</i> : int): bool	Function
Return true iff <i>i</i> is even, odd, equal to zero, positive or negative, respectively.	
abs (<i>i</i> : int): int	Function
Return the absolute value of <i>i</i> , that is, remove <i>i</i> 's sign.	
signum (<i>i</i> : int): int	Function
Return 1 if <i>i</i> is greater than zero, 0 if <i>i</i> is equal to zero and -1 if <i>i</i> is less than zero.	
pred (<i>i</i> : int): int	Function
succ (<i>i</i> : int): int	Function
Return the predecessor or successor of <i>i</i> , respectively.	
pow (<i>b</i> : int, <i>e</i> : int): int	Function
Return <i>b</i> raised to the power of <i>e</i> .	

5.3.2 longs module

This is a library module which exports long data type related functions.

min : long	Constant
min is the smallest representable long value.	
max : long	Constant
max is the largest representable long value.	
min (<i>x</i> : long, <i>y</i> : long): long	Function
Return the minimum of two long values.	
max (<i>x</i> : long, <i>y</i> : long): long	Function
Return the maximum of two long values.	
from_string (<i>s</i> : string): long	Function
Convert the string value <i>s</i> to the long number it represents. If <i>s</i> does not represent any long integer number, the return value is unspecified.	
to_string (<i>l</i> : long): string	Function
Convert the long number <i>l</i> to its string representation.	
from_int (<i>i</i> : int): long	Function
Convert the integer value <i>i</i> to a long value.	
to_int (<i>l</i> : long): int	Function
Convert the long value <i>l</i> to an integer value.	
from_real (<i>r</i> : real): long	Function
Convert the real number <i>r</i> to a long value, stripping off any decimal places.	

to_real (<i>l</i> : long): real	Function
Convert the long value <i>l</i> to a real value.	
even? (<i>l</i> : long): bool	Function
odd? (<i>l</i> : long): bool	Function
zero? (<i>l</i> : long): bool	Function
positive? (<i>l</i> : long): bool	Function
negative? (<i>l</i> : long): bool	Function
Return true iff <i>l</i> is even, odd, equal to zero, positive or negative, respectively.	
abs (<i>l</i> : long): long	Function
Return the absolute value of <i>l</i> , that is, remove <i>l</i> 's sign.	
signum (<i>l</i> : long): long	Function
Return 1L if <i>l</i> is greater than zero, 0L if <i>l</i> is equal to zero and -1L if <i>l</i> is less than zero.	
pred (<i>l</i> : long): long	Function
succ (<i>l</i> : long): long	Function
Return the predecessor or successor of <i>i</i> , respectively.	
pow (<i>b</i> : long, <i>e</i> : int): long	Function
Return <i>b</i> raised to the power of <i>e</i> .	

5.3.3 reals module

This is a library module which exports real data type related functions.

min (<i>x</i> : real, <i>y</i> : real): real	Function
Return the minimum of two real values.	
max (<i>x</i> : real, <i>y</i> : real): real	Function
Return the maximum of two real values.	
from_string (<i>s</i> : string): real	Function
Convert the string value <i>s</i> to the real number it represents. If <i>s</i> does not represent any integer number, the return value is unspecified.	
to_string (<i>r</i> : real): string	Function
Convert the real number <i>r</i> to its string representation.	
from_int (<i>i</i> : int): real	Function
Convert the integer value <i>i</i> to a real value.	
to_int (<i>r</i> : real): int	Function
Convert the real value <i>r</i> to an integer value. Decimal places are stripped.	
zero? (<i>r</i> : real): bool	Function
positive? (<i>r</i> : real): bool	Function
negative? (<i>r</i> : real): bool	Function
Return true iff <i>i</i> is equal to zero, positive or negative, respectively.	
abs (<i>r</i> : real): real	Function
Return the absolute value of <i>r</i> , that is, remove <i>r</i> 's sign.	
signum (<i>r</i> : real): real	Function
Return 1.0 if <i>r</i> is greater than zero, 0.0 if <i>r</i> is equal to zero and -1.0 if <i>r</i> is less than zero.	
pow (<i>a</i> : real, <i>b</i> : int): real	Function
Return <i>a</i> raised to the power of <i>b</i> .	

5.3.4 bools module

Library module for utility functions for boolean values.

to_string (<i>b</i> : bool): string	Function
Convert the boolean value <i>b</i> to a string. The result will be either the string "true" or the string "false".	
from_string (<i>s</i> : string): bool	Function
Convert a string to a boolean value. The two recognized strings are "true" and "false". Any other string will cause an unspecified value to be returned.	

5.3.5 chars module

Library module for character utilities.

min : char	Constant
max : char	Constant
EOF : char	Constant
EOF is the character which is returned by input functions when the end of an input file is reached.	
tab : char	Constant
Convenience variable containing the character '\t'.	
newline : char	Constant
Convenience variable containing the character '\n'.	
carriage_return : char	Constant
Convenience variable containing the character '\r'.	
blank : char	Constant
Convenience variable containing the character ' '.	
space : char	Constant
Convenience variable containing the character ' '.	
vtab : char	Constant
Convenience variable containing the character '\v'.	
backspace : char	Constant
Convenience variable containing the character '\b'.	
formfeed : char	Constant
Convenience variable containing the character '\f'.	
bell : char	Constant
Convenience variable containing the character '\a'.	
ord (<i>c</i> : char): int	Function
Return the character code of character <i>c</i> .	
chr (<i>i</i> : int): char	Function
Return the character which has character code <i>i</i> .	
to_string (<i>c</i> : char): string	Function
Convert the character <i>c</i> to a string of length one which only contains <i>c</i> .	

from_string (<i>s</i> : string): char	Function
Convert a string to a character value by simply extracting the first character. An exception will be raised if the string is empty.	
digit? (<i>c</i> : char): bool	Function
letter? (<i>c</i> : char): bool	Function
uppercase? (<i>c</i> : char): bool	Function
lowercase? (<i>c</i> : char): bool	Function
control? (<i>c</i> : char): bool	Function
punctuation? (<i>c</i> : char): bool	Function
letgit? (<i>c</i> : char): bool	Function
space? (<i>c</i> : char): bool	Function
whitespace? (<i>c</i> : char): bool	Function
printable? (<i>c</i> : char): bool	Function
Return true, if <i>c</i> is a digit, a letter, an uppercase letter or a lowercase letter, respectively.	
upcase (<i>c</i> : char): char	Function
If <i>c</i> is a lowercase letter, return its uppercase equivalent, otherwise, return <i>c</i> .	
downcase (<i>c</i> : char): char	Function
If <i>c</i> is an uppercase letter, return its lowercase equivalent, otherwise, return <i>c</i> .	
pred (<i>c</i> : char): char	Function
succ (<i>c</i> : char): char	Function
Return the predecessor or successor of <i>c</i> , respectively.	

5.3.6 strings module

Library module for string utilities.

length (<i>s</i> : string): int	Function
Return the number of characters in <i>s</i> .	
copy (<i>s</i> : string): string	Function
Return a freshly allocated string with the same length and contents as the given string <i>s</i> .	
substring (<i>s</i> : string, <i>from</i> : int, <i>to</i> : int): string	Function
Return a string containing the characters from <i>s</i> starting at index <i>from</i> (inclusive), up to <i>to</i> (exclusive).	
substring (<i>s</i> : string, <i>from</i> : int): string	Function
Similar to <code>substring(string, int, int)</code> , where the last argument defaults to the length of the string.	
append (<i>s1</i> : string, <i>s2</i> : string): string	Function
append (<i>s1</i> : string, <i>s2</i> : string, <i>s3</i> : string): string	Function
Return a string which is the concatenation of the argument strings.	
to_string (<i>b</i> : bool): string	Function
to_string (<i>i</i> : int): string	Function
to_string (<i>l</i> : long): string	Function
to_string (<i>r</i> : real): string	Function
to_string (<i>c</i> : char): string	Function
Return the string representation of the argument.	

- index** (*s*: string, *c*: char): int Function
 Return the smallest index of any appearance of character *c* in the string *s*. Return -1 if not found.
- rindex** (*s*: string, *c*: char): int Function
 Return the largest index of any appearance of character *c* in the string *s*. Return -1 if not found.
- indices** (*s*: string, *c*: char): list of int Function
 Return a list containing all indices of the appearances of character *c* in the string *s*. An empty list is returned if *c* does not appear in *s*.
- eq** (*s1*: string, *s2*: string): bool Function
 Return true if the two strings have the same length and contents, false otherwise.
- explode** (*s*: string): list of char Function
 Return a list of characters containing the characters of string *s*, in the same order.
- rexplode** (*s*: string): list of char Function
 Return a list of characters containing the characters of string *s*, in reversed order. This function is often convenient when constructing strings from character lists.
- implode** (*l*: list of char): string Function
 Return a string with the same length as the list *l*, and with the elements of *l* as string contents, in the same order.
- rimplode** (*l*: list of char): string Function
 Return a string with the same length as the list *l* and with the elements of *l* as string contents, but in reverse order. The same result would be obtained by calling
`strings.implode (lists.reverse (l));`
 but calling `rimplode` is more efficient.
- split** (*s*: string, *c*: char): list of string Function
 Split up the string *s* at all occurrences of the character *c* and return a list of the strings between these occurrences. Note that occurrences of *c* without any characters between will result in empty strings in the resulting list.
`strings.split ("root:x:0:0:root:/root:/bin/bash", ':')`
 \Rightarrow
`["root", "x", "0", "0", "root", "/root", "/bin/bash"]`
`strings.split (":x:0:0:root:./bin/bash", ':')`
 \Rightarrow
`["", "x", "0", "0", "root", "", "/bin/bash"]`
- replicate** (*elem*: char, *len*: int): string Function
 Return a string of length *len*, consisting of the character *elem*.
- rpadd** (*s*: string, *places*: int, *ch*: char): string Function
lpadd (*s*: string, *places*: int, *ch*: char): string Function
 Return a string of at least *places* characters which contains *s*, padded on the right (for `rpadd`) or left (for `lpadd`) with character *ch*.
- uppercase** (*s*: string): string Function
lowercase (*s*: string): string Function
 Return a newly allocated string containing the characters from *s* converted to upper case or lower case, respectively.

5.3.7 union module

This module exports an union type for the builtin data types.

union

Data type

Defined as:

```
datatype union =
  i(i: int) or
  l(l: long) or
  r(r: real) or
  b(b: bool) or
  c(c: char) or
  s(s: string)
```

This data type is intended to be used for functions which should be able to deal with all or some of the builtin data types.

5.3.8 strformat module

Library module for string formatting.

fmt (*fmt*: string, *args*: list of string): string Function

String formatting function. This is similar to the `sprintf()` function in C, but much more limited. *fmt* is expected to be a string with embedded formatting instruction. A formatting instruction is a % character followed by an *s* character. In the result string, each occurrence of a formatting instruction is replaced by the corresponding element in the argument list.

fmt (*fmt*: string, *s*: string): string Function

fmt (*fmt*: string, *s1*: string, *s2*: string): string Function

Convenience versions of the function above, to be used for one or two argument strings.

fmt (*fmt*: string, *args*: list of union.union): string Function

Generalized version of the formatting function. This accepts a list of `union.union` values, which can hold different data values. The supported formatting sequences are:

%s	Format a string.
%i, %d	Format an integer value.
%l	Format a long value.
%r	Format a real value.
%c	Format a character.
%b	Format a boolean value.

5.4 List utility modules

Turtle comes with a set of list utility functions, for example for constructing, investigating, searching or sorting.

5.4.1 lists module

The module file ‘`lists`’ exports list manipulation functions. It is a generic module with one module parameter, which is the type of the list elements. The name of this parameter is `A`.

empty? (<i>l</i> : list of A): bool	Function
Return true if <i>l</i> is the empty list, false otherwise.	
cons (<i>a</i> : A, <i>b</i> : list of A): list of A	Function
Cons the element <i>a</i> onto the head of list <i>b</i> .	
head (<i>l</i> : list of A): A	Function
Return the head of list <i>l</i> . An exception is raised if <i>l</i> is empty.	
tail (<i>l</i> : list of A): list of A	Function
Return the tail of list <i>l</i> . An exception is raised if <i>l</i> is empty.	
last (<i>l</i> : list of A): A	Function
Return the last element of list <i>l</i> . An exception is raised if <i>l</i> is empty.	
init (<i>l</i> : list of A): list of A	Function
Return the initial sequence of list <i>l</i> , excluding the last element. An exception is raised if <i>l</i> is empty.	
append (<i>a</i> : list of A, <i>b</i> : list of A): list of A	Function
Append the lists <i>a</i> and <i>b</i> .	
concat (<i>lists</i> : list of list of A): list of A	Function
Return a list which is the concatenation of the lists in <i>lists</i> .	
length (<i>a</i> : list of A): int	Function
Return the length of list <i>a</i> .	
reverse (<i>l</i> : list of A): list of A	Function
Return a list with the elements of <i>l</i> in reserved order.	
foreach (<i>p</i> : fun(A), <i>l</i> : list of A)	Function
Apply the procedure <i>p</i> to each element of the list <i>l</i> , in order. For a function which returns a list of the results of the function application, see module <code>listmap</code> (see Section 5.4.2 [listmap module], page 41).	
iota (<i>count</i> : int): list of int	Function
Return an integer list of <i>count</i> elements, with the number 0 to <i>count</i> -1 as the list elements.	
filter (<i>p</i> : fun(A): bool, <i>l</i> : list of A): list of A	Function
Return a list with all elements from <i>l</i> which satisfy predicate <i>p</i> , in the same order as in <i>l</i> .	
insert (<i>s</i> : A, <i>l</i> : list of A, <i>cmp</i> : fun(A, A): int): list of A	Function
Insert the element <i>s</i> into the list <i>l</i> , maintaining the order as defined by the comparison function <i>cmp</i> which takes two elements of type A and returns a value less than 0 if the first is smaller than the second, 0 if they are to be considered equal and a value greater than 0 if the second is greater.	
index (<i>elem</i> : A, <i>l</i> : list of A, <i>cmp</i> : fun(A, A): int): int	Function
Return the index of element <i>elem</i> in list <i>l</i> , according to the comparison function <i>cmp</i> . Return -1 if <i>elem</i> does not appear in <i>l</i> .	

indices (<i>elem</i> : A, <i>l</i> : list of A, <i>cmp</i> : fun(A, A): int): list of int	Function
Return a list of the indices of all appearances of <i>elem</i> in <i>l</i> , according to the comparison function <i>cmp</i> . Return the empty list if <i>elem</i> does not appear in <i>l</i> .	
replicate (<i>elem</i> : A, <i>len</i> : int): list of A	Function
Create a list of length <i>l</i> , where all list elements are initialized to <i>elem</i> .	
copy (<i>l</i> : list of A): list of A	Function
Create a copy of the list <i>l</i> . Note that only the spine of the list is copied, not the elements.	
take (<i>l</i> : list of A, <i>n</i> : int): list of A	Function
Take the first <i>n</i> elements from the list <i>l</i> , dropping the following elements.	
drop (<i>l</i> : list of A, <i>n</i> : int): list of A	Function
Drop the first <i>n</i> elements from list <i>l</i> and return the remaining list.	

5.4.2 listmap module

The module file ‘listmap’ exports higher-order functions for iterating over lists. It is a generic module with two module parameters, which are the types of the list elements. The name of the first parameter is *A* and denotes the element type of the input lists, the name of the second is *B* and stands for the element type of the output lists of `map`.

map (<i>f</i> : fun(A): B, <i>l</i> : list of A): list of B	Function
Apply the function <i>f</i> to every element of <i>l</i> , return a list containing the results of the function applications. The order in which <i>f</i> is applied to the list elements is not specified.	

5.4.3 listfold module

The module file ‘listfold’ implements the functions `foldl` and `foldr` for folding functions over lists.

foldl (<i>f</i> : fun(A, A): A, <i>l</i> : list of A): A	Function
Fold the function <i>f</i> from left to right over the list <i>l</i> and return the result.	
<pre> fun sub(x: int, y: int): int return x - y; end; foldl (sub, [4, 3, 1]) ⇒ (4 - 3) - 1 ⇒ 0 </pre>	
foldr (<i>f</i> : fun(A, A): A, <i>l</i> : list of A): A	Function
Fold the function <i>f</i> from right to left over the list <i>l</i> and return the result.	
<pre> fun sub(x: int, y: int): int return x - y; end; foldr (sub, [4, 3, 1]) ⇒ 4 - (3 - 1) ⇒ 2 </pre>	

5.4.4 listreduce module

The module file ‘listreduce’ implements the functions `reducel` and `reducer` for reductant functions over lists.

reducel (*f*: fun(from, to): to, *init*: to, *l*: list of from): to Function

Reduce the list *l* with function *f*, bracketing on the left.

```

fun sub (x: int, y: int): int
return x - y;
end;
reducel (sub, 12, [2, 5, 3])
⇒
3 - (5 - (2 - 12))
⇒
-12

```

reducer (*f*: fun(from, to): to, *init*: to, *l*: list of from): to Function

Reduce the list *l* with function *f*, bracketing on the right.

```

fun sub (x: int, y: int): int
return x - y;
end;
reducel (sub, 12, [2, 5, 3])
⇒
2 - (5 - (3 - 12))
⇒
-12

```

5.4.5 listzip module

Functions for combining two lists into one element-wise, and the reverse.

zip (*f*: fun(from1, from2): to, *l1*: list of from1, *l2*: list of from2): list of to Function

Zip two lists of equal length together by applying the function *f* to the corresponding elements of the lists and forming the result lists from the function result(s).

unzip (*f*: fun(to): (from1, from2), *l*: list of to): (list of from1, list of from2) Function

Decompose the list *l* by applying the function *f* to the successive list elements and returning the two lists of the function results.

5.4.6 listindex module

Some utility functions for lists which work on indices into the list.

nth (*l*: list of A, *idx*: int): A Function

Return the element at index *idx* in list *l*.

pos (*p*: fun(A): bool, *l*: list of A): int Function

Return the index of the first element in list *l* which satisfies predicate *p*. Return -1 if no element satisfies *p*.

slice (*l*: list of A, *start*: int, *ende*: int): list of A Function

Return the sublist of *l* which starts at index *start* (inclusive) and ends at index *ende* (exclusive).

5.4.7 listsearch module

Function for searching in lists of type

`list of A`

where A must be instantiated when importing this module.

The comparison function *cmp* is used for comparing elements of the input arrays. *cmp* is expected to return a value less than 0 if the first argument is to be considered smaller than the second, a value greater than 0 if it is greater, and exactly 0 if the two arguments are equivalent.

lsearch (*l*: list of A, *elem*: A, *cmp*: fun(A, A): int): list of A Function
 Search linearly through the list *l*, until an element equal to *elem* is found. Use *cmp* for comparing the elements of the list to *elem*. Return the first list cell whose head is equal to *elem*, or `null` if not found.

5.4.8 listsort module

Function for sorting lists of type

`list of A`

where A must be instantiated when importing this module.

The comparison function *cmp* is used for comparing elements of the input arrays. *cmp* is expected to return a value less than 0 if the first argument is to be considered smaller than the second, a value greater than 0 if it is greater, and exactly 0 if the two arguments are equivalent.

sort (*a*: list of A, *cmp*: fun(A, A): int): list of A Function
 Sort the list *a*, using *cmp* as the comparison function. The parameter *a* is not modified for performing the sort, instead a freshly allocated list is returned.

5.5 Array utility modules

Several modules for manipulating arrays are included in the standard library. They are structured similarly to the modules for list handling.

5.5.1 arrays module

Utility functions for handling objects of type

`array of A`

where A must be instantiated when importing this module.

length (*a*: array of A): int Function
 Return the number of elements of array 'a'.

foreach (*p*: fun(A), *a*: array of A) Function
 Apply the procedure *p* to each element of the array *a*, in order.

copy (*a*: array of A): array of A Function
 Create a copy of array *a*. Note that only the array holding the elements is copied, not the elements themselves. The result is the same as *a* if *a* is empty.

reverse (*a*: array of A): array of A Function
 Return a new array which contains the elements of *a*, but in reverse order. The result is the same as *a* if *a* is empty.

replicate (*elem*: A, *len*: int): array of A Function
 Create an array of *len* elements, where each element is initialized to *elem*.

5.5.2 arraymap module

Utility function for applying functions to arrays of type
array of *A*.

Map returns values of type
array of *B*.

A and *B* are type variables which must be instantiated when importing this module.

map (*f*: fun(*A*): *B*, *l*: array of *A*): array of *B* Function
Apply the function *f* to every element of the array *a*, return an array containing the results of the function applications. The order in which *f* is applied to the array elements is not specified.

5.5.3 arraysearch module

This module exports a functions for sorting objects of type
array of *A*

where *A* must be instantiated when importing this module.

The comparison function **cmp** is used for comparing elements of the input arrays. **cmp** is expected to return a value less than 0 if the first argument is to be considered smaller than the second, a value greater than 0 if it is greater, and exactly 0 if the two arguments are equivalent.

lsearch (*a*: array of *A*, *elem*: *A*, *cmp*: fun(*A*, *A*): int): int Function
Search linearly through the array *a*, until an element equal to *elem* is found. Use *cmp* for comparing the elements of the array to *elem*. Return the index of the first occurrence if found; return -1 otherwise.

Note that if *elem* appear more than once in the array, the index of the first occurrence is returned.

bsearch (*a*: array of *A*, *elem*: *A*, *cmp*: fun(*A*, *A*): int): int Function
Binary search. Search in the array *a* for an element equal to *elem*, using *cmp* as the comparison function. Return the index of the element if found, return -1 otherwise.

The array *a* must be sorted according to the comparison function *cmp*, otherwise the search will most probably fail, even if *elem* does appear in *a*.

Note that if *elem* appear more than once in the array, it is not specified the index of which will be returned.

5.5.4 arraysort module

This module exports a functions for sorting objects of type
array of *A*

where *A* must be instantiated when importing this module.

The comparison function **cmp** is used for comparing elements of the input arrays. **cmp** is expected to return a value less than 0 if the first argument is to be considered smaller than the second, a value greater than 0 if it is greater, and exactly 0 if the two arguments are equivalent.

sort (*a*: array of *A*, *cmp*: fun(*A*, *A*): int) Function
Sort the array *a*, using *cmp* as the comparison function. The parameter *a* is modified for performing the sort.

5.6 Tuple utility modules

Two modules for handling common tuple types are also in the distribution, one for handling pairs of two data types, and one for handling triples of three data types.

5.6.1 pairs module

Utility functions for tuples of type

(A, B)

where A and B must be instantiated when importing this module.

pair

Data type

Defined as:

```
datatype pair<A, B> =
  pair(first: A, second: B)
```

This datatype wraps a pair of values in an user-defined data type.

unpair (p : pair): (A, B)

Function

pair (p : (A, B)): pair

Function

These are conversion functions between the user-defined **pair** data type and 2-tuples.

first (p : (A, B)): A

Function

second (p : (A, B)): B

Function

Return the first or second component of the argument tuple, respectively.

5.6.2 triples module

Utility functions for tuples of type

(A, B, C)

where A , B and C must be instantiated when importing this module.

triple

Data type

Defined as:

```
datatype triple<A, B> =
  triple(first: A, second: B, third: C)
```

This datatype wraps a 3-tuples of values in an user-defined data type.

untriple (t : triple): (A, B, C)

Function

triple (t : (A, B, C)): triple

Function

These are conversion functions between the user-defined **triple** data type and 3-tuples.

first (p : (A, B, C)): A

Function

second (p : (A, B, C)): B

Function

third (p : (A, B, C)): C

Function

Return the first, second or third component of the argument tuple, respectively.

5.7 Low level modules

Some of the modules in the standard library are very low-level. These modules are either not convenient to use, or are machine or system dependent, so it is a good idea to avoid using them as far as possible, and rather use the functionality implemented by the higher-level modules in the previous sections.

A lot of functions in the higher-level modules are implemented in terms of the low-level ones, and sometimes access to low-level features is useful. That is the reason why these modules are documented here even though their use is discouraged.

5.7.1 core module

This file defines some useful low-level functions, but does not implement all of them. For some of the functions only the declaration is given, and the implementation is written in C in the file `core.t.i`.

The module `core` is a very basic library module. Very low-level functions for input and output are provided. The user should not use these functions directly, but use one of the higher-level modules like `io` instead.

write_char (<i>fd</i> : int, <i>c</i> : char)	Function
Write the character <i>c</i> to the file descriptor <i>fd</i> .	
read_char (<i>fd</i> : int): char	Function
Read a character from file descriptor <i>fd</i> . On end-of-file, the constant <code>chars.EOF</code> is returned.	
real_to_string (<i>r</i> : real): string	Function
Convert the real number <i>r</i> to its string representation.	
string_to_real (<i>s</i> : string): real	Function
Convert the string value <i>s</i> to a real value. If <i>s</i> is not a valid real number representation, the returned value is undefined.	
chr (<i>i</i> : int): char	Function
Return the character with character code <i>i</i> .	
ord (<i>c</i> : char): int	Function
Return the character code of the character <i>c</i> .	
int_to_real (<i>i</i> : int): real	Function
Convert the integer value <i>i</i> to a real value.	
real_to_int (<i>r</i> : real): int	Function
Convert the real value <i>r</i> to an integer value, stripping off any decimal places.	

5.8 Subsystem sys

The subsystem `sys` contains some system-dependant modules which access operating system specific functions.

All functions and variables in these modules are close to the underlying operating system calls and C library functions of the same name, and the semantics are the same as far as possible. Only some minor changes to the semantics have been made in order to map the different data types in Turtle and C onto each other.

5.8.1 sys.files module

Module for handling files.

stdin : int	Constant
stdout : int	Constant
stderr : int	Constant
File descriptors for standard input, output and error..	
open (<i>fname</i> : string): int	Function
Open the existing file named <i>fname</i> for reading and return a file descriptor; or return -1 on error.	

create (<i>fname</i> : string): int	Function
Create a new file named <i>fname</i> and open it for writing and return a file descriptor; or return -1 on error. An existing file named <i>fname</i> will be overwritten, so use with caution.	
close (<i>fd</i> : int): int	Function
Close the file associated with file descriptor <i>fd</i> , and return 0 on success or -1 on error.	
close (<i>fd</i> : int)	Function
Close the file associated with file descriptor <i>fd</i> . Ignore any errors.	
write (<i>fd</i> : int, <i>b</i> : internal.binary.binary, <i>len</i> : int): int	Function
Write <i>len</i> bytes from the byte array <i>b</i> to the file descriptor <i>fd</i> , starting at offset 0 of the byte array. Return the number of bytes actually written, or -1 if an error occurs. The variable <code>sys.errno.errno</code> is set accordingly.	
read (<i>fd</i> : int, <i>b</i> : internal.binary.binary, <i>len</i> : int): int	Function
Read <i>len</i> bytes from the file descriptor <i>fd</i> into the byte array <i>b</i> . Return the number of bytes read, or -1 if an error occurs. The variable <code>sys.errno.errno</code> is set accordingly.	
unlink (<i>filename</i> : string): int	Function
Delete a name from the filesystem. If that name was the alst linkt to a file and no processes have the file open the file is deleted and the space it was using is made available for reuse.	
On success, zero is returned. On error, -1 is returned and the variable <code>sys.errno.errno</code> is set appropriately.	

5.8.2 sys.dirs module

Module for handling directories. The exported data type `dirs` serves as a handle for directories and must be obtained by calling the function `opendir`. Calling `readdir` repeatedly on a valid `dir` value will return all entries of a directory, and the directory stream can then be closed with `closedir` or reset to the beginning with `rewinddir`.

dir	Data type
Defined as:	
<pre>datatype dir = adir</pre>	
Directory handle for use with the directory functions below.	
opendir (<i>name</i> : string): dir	Function
Open a directory stream corresponding with the directory called <i>name</i> , and return a handle for that stream. The stream is positioned at the first entry in the directory. Return a handle for the opened stream on success, or <code>null</code> if an error occurs. The variable <code>sys.errno.errno</code> will be set accordingly.	
readdir (<i>d</i> : dir): string	Function
Return a string representing the next directory entry in the directory stream <i>d</i> . Return <code>null</code> if the end of the stream is reached or an error occurs. <code>sys.errno.errno</code> will be set accordingly.	
closedir (<i>d</i> : dir): int	Function
Close the directory stream associated with the handle <i>d</i> . The stream descriptor <i>d</i> cannot be used anymore after this call. Return 0 on success or -1 on failure, and set <code>sys.errno.errno</code> accordingly.	
rewinddir (<i>d</i> : dir)	Function
Reset the position of the directory stream <i>d</i> to the beginning of the directory.	

5.8.3 sys.net module

Module for network programming.

Currently, only IPv4 is supported, and only the most basic operations are provided.

PF_UNSPEC : int Constant
PF_UNIX : int Constant
PF_LOCAL : int Constant
PF_INET : int Constant

These constants are to be used as the *domain* parameter to the `socket` function. Note that currently only IPv4 (`PF_INET`) sockets are supported.

SOCK_STREAM : int Constant
SOCK_DGRAM : int Constant
SOCK_RAW : int Constant

These are the socket type constants for calls to the `socket` function. Only stream sockets (`SOCK_STREAM`) have been tested yet.

socket (*domain*: int, *typ*: int, *protocol*: int): int Function

Create an endpoint for communication and return a descriptor.

The *domain* parameter specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined as the `PF_*` constants.

The socket has the indicated type *typ*, which specifies the communication semantics. The currently defined types are the `SOCK_*` constants.

-1 is returned if an error occurs and `sys.errno.errno` is set accordingly, otherwise the return value is a descriptor referencing the socket.

sockaddr Data type

Defined as:

```
datatype sockaddr =
  inet(port: int, addr: internal.binary.binary)
```

Data type for specifying network addresses. Currently only IPv4 is supported.

sockaddr_addr (*addr*: sockaddr): internal.binary.binary Function

sockaddr_port (*addr*: sockaddr): int Function

Return the IP number or the port of the socket address *addr*, respectively.

bind (*sockfd*: int, *addr*: sockaddr): int Function

Give the socket *sockfd* the local address *addr*. This is necessary for a stream socket may receive connections with `accept`.

connect (*sockfd*: int, *serv_addr*: sockaddr): int Function

Establish a connection for socket descriptor *sockfd* to the server specified by *serv_addr*. The return value is -1 if an error occurs and zero on success.

listen (*sockfd*: int, *backlog*: int): int Function

Before accepting connections with `accept`, the willingness to accept incoming connections and a queue limit for incoming connections are specified with `listen`.

accept (*sockfd*: int): (int, sockaddr) Function

Accept a client connection on the given socket *sockfd*. Return a pair of a socket descriptor for the connection to the client, and the address of the client.

inetaddr (*port*: int, *b0*: int, *b1*: int, *b2*: int, *b3*: int): sockaddr Function
inetaddr (*port*: int): sockaddr Function
inetaddr (*port*: int, *b*: internal.binary.binary): sockaddr Function

Create an internet address structure with a given *port* and the given bytes of the internet address. The bytes are to be passed highest-order byte first, for example:

```
"127.0.0.1"
⇒
127, 0, 0, 1
```

The second version of the function will set the address to zero and can be used when binding an address for a server socket where the address should be the default.

gethostbyname (*name*: string): internal.binary.binary Function

Return the IPv4 address of host *name*. *name* must be either a valid host name or an internet address in dotted decimal notation.

The return value is a byte array representing the internet address or `null`, if the host name cannot be resolved.

5.8.4 sys.times module

Module for handling times.

tm Data type

Defined as:

```
datatype tm =
  tm(sec: int, min: int, hour: int, mday: int, mon: int, year: int, wday: int,
```

This data type represents date and time information.

gmtime (*t*: long): tm Function

Return a `tm` structure representing the current time in Universal Coordinated Time (UTC).

localtime (*t*: long): tm Function

Return a `tm` structure representing the current time in local time.

time (): long Function

Return the current time in seconds since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

asctime (*tm*: tm): string Function

Return a string representation of the time *tm* or *t*, respectively.

ctime (*t*: long): string Function

5.8.5 sys.users module

Module for accessing user information.

getuid (): int Function

geteuid (): int Function

`getuid` returns the real user ID of the calling process.

`geteuid` returns the effective user ID of the current process. The effective ID corresponds to the set ID bit on the file being executed.

- passwd** Data type
 Defined as:
- ```
datatype passwd =
 passwd(name: string, passwd: string, uid: int, gid: int, gecost: string, dir: string)
```
- group** Data type  
 Defined as:
- ```
datatype group =
  group(name: string, passwd: string, gid: int, members: array of string)
```
- getpwnam** (*name*: string): passwd Function
getpwuid (*uid*: int): passwd Function
 Return the password file structure for the user with the given user name or user id, respectively. If the user name or id is not valid on the system the program runs on, null is returned.
- getpwent** (): passwd Function
 Return the next entry from the password file, as a value of the data type `sys.users.passwd`. If called for the first time, the first entry will be returned, then successive entries until the end of the user data base is reached. The end of the file is indicated by returning null.
- setpwent** () Function
 Reset the read pointer for the user data base so that the next call to `getpwent` will return the first entry.
- endpwent** () Function
 Close the password file. Use this function when you are ready with the user data base.
- getgrnam** (*name*: string): group Function
getgrgid (*uid*: int): group Function
 Return the group file structure for the group with the given group name or group id, respectively. If the group name or id is not valid on the system the program runs on, null is returned.
- getgrent** (): group Function
 Return the next entry from the group file, as a value of the data type `sys.users.group`. If called for the first time, the first entry will be returned, then successive entries until the end of the group data base is reached. The end of the file is indicated by returning null.
- setgrent** () Function
 Reset the read pointer for the group data base so that the next call to `getgrent` will return the first entry.
- endgrent** () Function
 Close the group file. Use this function when you are ready with the group data base.
- getlogin** (): string Function
 Return a string containing the name of the user logged in on the controlling terminal of the process, or an empty string if this information cannot be determined.

5.8.6 sys.procs module

Module for accessing process information and dealing with processes.

- getpid ()**: int Function
 Return the process identifier of the current process.
- getppid ()**: int Function
 Return the process identifier of the parent of the current process.
- sleep (sec: int)**: int Function
sleep (sec: int) Function
 Make the current process sleep until *sec* seconds have elapsed or a signal arrives which is not ignored.
 The first version returns zero if the requested time has elapsed, or the number of seconds left to sleep. The second version does not return anything.
- exit (status: int)** Function
 Terminate the current process normally and return the value of *status* to the parent.
- kill (pid: int, sig: int)**: int Function
kill (pid: int, sig: int) Function
kill can be used to send any signal to any process group or process.
 If *pid* is positive, then signal *sig* is sent to *pid*. If *pid* equals 0, then *sig* is sent to every process in the process group of the current process. If *pid* equals -1, then *sig* is sent to every process except for the first one, from higher numbers in the process table to lower. If *pid* is less than -1, then *sig* is sent to every process in the process group *-pid*.
 If *sig* is 0, then no signal is sent, but error checking is still performed.
 On success, zero is returned, On error, -1 is returned and **errno** is set appropriately. The second version of the **kill** function does not return anything and ignores any errors.
- fork ()**: int Function
 Create a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.
 On success, the process identifier of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, -1 will be returned in the parent's context, no child process will be created, and the variable **errno.errno** will be set appropriately.
- wait ()**: (int, int) Function
 Suspend execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.
 The return value is a pair of the process ID of the exited child (or -1 on error) and the status of the exited child.
- WNOHANG** : int Constant
WUNTRACED : int Constant
 Constants to be used as the options argument to **waitpid**.

waitpid (*pid*: int, *options*: int): int Function

Suspend execution of the current process until a child as specified by the *pid* argument, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child as requested by *pid* has already exited by the time of the call (a so-called "zombie" process", the function returns immediately. Any system resources used by the child are freed.

The value of *pid* can be one of

- < -1 which means to wait for any child process whose process group ID is equal to the absolute value of *pid*.
- 1 which means to wait for any child process; this is the same behaviour which `wait` exhibits.
- 0 which means to wait for any child process whose process group ID is equal to that of the calling process.
- > 0 which means to wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG which means to return immediately if no child has exited.

WUNTRACED which means to also return for children which are stopped, and whose status has not been reported.

The return value is a pair of the process ID of the exited child (or -1 on error) and the status of the exited child.

If **WNOHANG** was given as an option, and no child has exited, 0 is returned as the process ID.

WIFEXITED (*status*: int): bool Function
Return `true`, if the status code indicates that the process has exited normally.

WEXITSTATUS (*status*: int): int Function
Return the exit status of the exited process. This may only be called if **WIFEXITED** returned true for *status*.

WIFSIGNALED (*status*: int): bool Function
Return `true`, if the status code indicates that the process was terminated by a signal.

WTERMSIG (*status*: int): int Function
Return the number of the signal which terminated the process. This may only be called if **WIFSIGNALED** returned true for *status*.

WIFSTOPPED (*status*: int): bool Function
Return `true`, if the status code indicates that the process was stopped by a signal.

WSTOPSIG (*status*: int): int Function
Return the number of the signal which stopped the process. This may only be called if **WIFSTOPPED** returned true for *status*.

execve (<i>filename</i> : string, <i>argv</i> : array of string, <i>envp</i> : array of string): int	Function
execve (<i>filename</i> : string, <i>argv</i> : array of string, <i>envp</i> : array of string)	Function
execve (<i>filename</i> : string, <i>argv</i> : list of string, <i>envp</i> : list of string): int	Function
execve (<i>filename</i> : string, <i>argv</i> : list of string, <i>envp</i> : list of string)	Function
execve (<i>filename</i> : string, <i>argv</i> : array of string): int	Function
execve (<i>filename</i> : string, <i>argv</i> : array of string)	Function
execve (<i>filename</i> : string, <i>argv</i> : list of string): int	Function
execve (<i>filename</i> : string, <i>argv</i> : list of string)	Function

Execute the program called *filename*. *argv* is an array or list of argument strings passed to the new program. *envp* is an array of string, conventionally of the form **key=value**, which are passed as environment to the new program.

Normally these functions do not return, on error, the value -1 is returned and the appropriate error code is placed in the variable `sys.errno.errno`.

getenv (<i>varname</i> : string): string	Function
Return the value of the environment variable called <i>varname</i>	

5.8.7 sys.errno module

Module for accessing operating system errors.

errno : int	Variable
The variable <i>errno</i> contains the result code set by the last operating system call. A value of 0 means success, all other values indicate failure. The value can be translated to a readable error message using the <code>strerror</code> function below.	

strerror (<i>errnum</i> : int): string	Function
Return a string describing the error code passed in the argument <i>errnum</i> .	

5.8.8 sys.signal module

Module for installing a signal function, that is a function which gets called whenever the process receives an operating system signal.

In Turtle, signals are handled synchronously, that means that a Turtle process repeatedly checks whether a signal has arrived and then calls the signal handler function.

SIGHUP : int	Variable
SIGINT : int	Variable
SIGQUIT : int	Variable
SIGILL : int	Variable
SIGABRT : int	Variable
SIGFPE : int	Variable
SIGKILL : int	Variable
SIGSEGV : int	Variable
SIGPIPE : int	Variable
SIGALRM : int	Variable
SIGTERM : int	Variable
SIGUSR1 : int	Variable
SIGUSR2 : int	Variable
SIGCHLD : int	Variable
SIGCONT : int	Variable
SIGSTOP : int	Variable
SIGTSTP : int	Variable
SIGTTIN : int	Variable
SIGTTOU : int	Variable

These are the signals defined in POSIX.1.

signal (<i>no</i> : int, <i>handler</i> : fun(int))	Function
---	----------

Install a signal handler for signal number *no*. Whenever a signal is received, the process will stop at the next safe point and call the handler function. with the signal number as the argument.

5.9 Subsystem internal

Some of the modules in the standard library are very low-level. These modules are either not convenient to use, or are machine or system dependent, so it is a good idea to avoid using them as far as possible, and rather use the functionality implemented by the higher-level modules in the previous sections.

A lot of functions in the higher-level modules are implemented in terms of the low-level ones, and sometimes access to low-level features is useful. That is the reason why these modules are documented here even though their use is discouraged.

5.9.1 internal.version module

Low-level module for version information.

version (): string	Function
---------------------------	----------

Return the version number of the running Turtle runtime in string form, e.g "0.1.1".

5.9.2 internal.random module

Low-level module for random numbers.

srand (<i>seed</i> : int)	Function
-----------------------------------	----------

Set the seed of the random number generator to *seed*.

rand (): int	Function
---------------------	----------

Return a random number in the range 0 to `ints.max`.

5.9.3 `internal.stats` module

Low-level module for interfacing to the runtime system statistics.

<code>dispatch_call_count ()</code>	: int	Function
<code>direct_call_count ()</code>	: int	Function
<code>local_call_count ()</code>	: int	Function
<code>closure_call_count ()</code>	: int	Function
<code>gc_checks ()</code>	: int	Function
<code>gc_calls ()</code>	: int	Function
<code>allocations ()</code>	: int	Function
<code>alloced_words ()</code>	: int	Function
<code>forwarded_words ()</code>	: int	Function
<code>save_cont_count ()</code>	: int	Function
<code>restore_cont_count ()</code>	: int	Function
<code>total_gc_time ()</code>	: int	Function
<code>min_gc_time ()</code>	: int	Function
<code>max_gc_time ()</code>	: int	Function

These functions deliver some statistics gathered by the runtime system.

5.9.4 `internal.gc` module

Low-level module for interfacing to the garbage collector.

<code>gc_calls ()</code>	: int	Function
Return the number of heap garbage collections since the program started.		
<code>gc_checks ()</code>	: int	Function
Return the number of heap overflow checks since the program started.		
<code>garbage_collect ()</code>		Function
Force a garbage collection.		

5.9.5 `internal.ex` module

Low-level module for raising and handling exceptions. Do not use this module directly, rather use the standard library module `exception` (see Section 5.1.14 [exceptions module], page 30).

<code>raise (s: string)</code>		Function
Raise an exception with name <i>s</i> .		
<code>handle (thunk: fun(): (), handler: fun(string))</code>		Function
Call the function <i>thunk</i> . If any exception is raised while <i>thunk</i> is running, the function <i>handler</i> will be called with the exception name as the only argument. When <i>handler</i> returns, it will return to the caller of <code>ex.handle</code> , in the same way as the call would return when <i>thunk</i> was returning without an exception.		
<code>null_pointer_exception ()</code>	: string	Function
<code>out_of_range_exception ()</code>	: string	Function
<code>subscript_exception ()</code>	: string	Function
<code>wrong_variant_exception ()</code>	: string	Function
<code>require_exception ()</code>	: string	Function

The return value of these functions is the corresponding exception name, which is the same that would be used if the illegal operation would be performed.

That means that

```
internal.ex.raise (internal.ex.subscript_ex ())  
has the same effect as  
var s: string;  
s[-1] := 'a'
```

5.9.6 internal.timeout module

Module for installing a timeout function, that is a function which gets called repeatedly in some interval.

set (*handler*: fun()) Function
Register *handler* as the timeout function for the Turtle runtime. The timeout function gets called repeatedly, but there is no guarantee on how often it will be called.
On a 800Mhz AMD Duron(tm) processor, the function is called about every 0.1 seconds.

clear () Function
Remove the current timeout function, so that no function will be called until another handler is registered with the function **set** above.

5.9.7 internal.limits module

Low-level module defining certain system limits.

There are no exports yet, but that may change in the future.

Glossary

basic constraint

A basic constraint is a linear equation, which may contain constrainable and unconstrainable variables.

constrainable variable

A constrainable variable is a variable whose value can be determined by a constraint. Unconstrainable variables can only be changed by assignment.

constraint A constraint is a condition which specifies the properties of the values a solution to a certain problem must have.

declarative

With declarative programming, we name the programming style where the computer is told which properties a problem's solutions must have, but not how to compute the solution. The compiler and runtime system of the language implementation is responsible for determining an efficient way to compute the solution.

imperative

Imperative programming means the programming style where the computer is explicitly told which steps to perform when and in which order, to find the solution to a problem. Imperative programs are rather low-level, compared to declarative programs.

module

A module is a collection of functions, variables, constraints and type declarations, which can be compiled independently of other modules. A collection of modules forms a program.

variable

A variable is a storage location. The contents of variables can be fetched and new values can be stored into variables, using assignment. In Turtle, the values of variables can also be determined by using constraints on constrainable variables.

Index

A

abs 34, 35
 accept 48
 acos 25
 allocations 55
 allocated_words 55
 append 37, 40
 Array data type 19
 array modules 43
 array utilities 43
 arraymap (Module) 44
 arrays (Module) 43
 arraysearch (Module) 44
 arraysort (Module) 44
 asctime 49
 asin 25
 atan 25

B

backspace 36
 basename 31
 Basic data types 17
 bell 36
 binary (Module) 30
 binary_get 30
 binary_set 30
 binary_size 30
 bind 48
 bintree (Module) 29
 bla 25
 blank 36
 Booleans 18
 bools (Module) 36
 bsearch 44
 bstrees (Module) 29

C

carriage_return 36
 Characters 18
 chars (Module) 36
 chr 36, 46
 clear 56
 close 33, 47
 closedir 47
 closure_call_count 55
 cmdline (Module) 27
 cmp 26
 compare (Module) 26
 compose 26
 compose (Module) 26

Compound data types 19
 concat 40
 connect 48
 cons 40
 constrainable variable 23
 constraint statements 23
 constraint store 23
 constraints 23
 constructor 21
 control? 37
 copy 29, 37, 41, 43
 core (Module) 46
 cos 25
 create 33, 47
 ctime 49

D

data type related modules 33
 Data Types 17
 delete 28
 digit? 37
 direct_call_count 55
 dirname 31
 discriminator 21
 dispatch_call_count 55
 downcase 37, 38
 drop 41

E

empty? 40
 endgrent 50
 endpwent 50
 EOF 36
 eq 38
 errno 53
 error 31
 even? 34, 35
 exceptions (Module) 30
 execve 52, 53
 exit 51
 explode 38

F

filenames (Module) 31
 filter 40
 find 29
 first 45
 flush 33
 fmt 39
 foldl 41

foldr..... 41
 foreach..... 40, 43
 fork..... 51
 formfeed..... 36
 forwarded_words..... 55
 from_int..... 34, 35
 from_real..... 34
 from_string..... 30, 33, 34, 35, 36, 37
 Function data type..... 20

G

garbage_collect..... 55
 gc_calls..... 55
 gc_checks..... 55
 General modules..... 25
 get..... 32, 33
 getenv..... 53
 geteuid..... 49
 getgrent..... 50
 getgrgid..... 50
 getgrnam..... 50
 gethostbyname..... 49
 getlogin..... 50
 getopt..... 27
 getpid..... 51
 getppid..... 51
 getpwent..... 50
 getpwnam..... 50
 getpwuid..... 50
 getuid..... 49
 gmtime..... 49

H

handle..... 30, 55
 hash..... 27
 hash (Module)..... 27
 hashtable (Module)..... 28
 head..... 40

I

id..... 26
 identity (Module)..... 26
 implode..... 38
 index..... 38, 40
 indices..... 38, 41
 inetaddr..... 49
 init..... 40
 inorder..... 28
 input..... 31
 Input and output modules..... 31
 insert..... 28, 29, 40
 int_to_real..... 46
 Integers..... 17

internal.ex (Module)..... 55
 internal.gc (Module)..... 55
 internal.limits (Module)..... 56
 internal.random (Module)..... 54
 internal.stats (Module)..... 55
 internal.timeout (Module)..... 56
 internal.version (Module)..... 54
 ints (Module)..... 33
 io (Module)..... 31
 iota..... 40

K

kill..... 51

L

last..... 40
 length..... 37, 40, 43
 letgit?..... 37
 letter?..... 37
 List data type..... 20
 List modules..... 39
 List utilities..... 39
 listen..... 48
 listfold (Module)..... 41
 listindex (Module)..... 42
 listmap (Module)..... 41
 listreduce (Module)..... 42
 lists (Module)..... 40
 listsearch (Module)..... 43
 listsort (Module)..... 43
 listzip (Module)..... 42
 local_call_count..... 55
 localtime..... 49
 Longs..... 17
 longs (Module)..... 34
 lookup..... 28
 low level modules..... 45
 lowercase?..... 37
 lpad..... 38
 lsearch..... 43, 44

M

make..... 28
 make_binary..... 30
 map..... 41, 44
 math (Module)..... 25
 max..... 33, 34, 35, 36
 max_gc_time..... 55
 min..... 33, 34, 35, 36
 min_gc_time..... 55

N

negative?..... 34, 35

newline 36
 nl 32
 node 28
 nth 42
 null_pointer_ex 30
 null_pointer_exception 55

O

odd? 34, 35
 open 33, 46
 opendir 47
 option (Module) 26
 ord 36, 46
 out_of_range_ex 30
 out_of_range_exception 55
 output 31

P

pair 45
 pairs (Module) 45
 path_seperator 31
 PF_INET 48
 PF_LOCAL 48
 PF_UNIX 48
 PF_UNSPEC 48
 pi 25
 pos 42
 positive? 34, 35
 postorder 28
 pow 34, 35
 pred 34, 35, 37
 preorder 28
 printable? 37
 punctuation? 37
 put 31, 32
 putln 32

R

raise 30, 55
 rand 25, 54
 random (Module) 25
 read 47
 read_char 46
 readdir 47
 real_to_int 46
 real_to_string 46
 Reals 18
 reals (Module) 35
 reducel 42
 reducer 42
 replicate 38, 41, 43
 require_ex 30
 require_exception 55

restore_cont_count 55
 reverse 40, 43
 rewinddir 47
 rexplode 38
 rimplode 38
 rindex 38
 rpad 38

S

save_cont_count 55
 search 29
 second 45
 seed 25
 selector 21
 set 56
 setgrent 50
 setpwent 50
 setter 21
 SIGABRT 54
 SIGALRM 54
 SIGCHLD 54
 SIGCONT 54
 SIGFPE 54
 SIGHUP 53
 SIGILL 54
 SIGINT 54
 SIGKILL 54
 signal 54
 signum 34, 35
 SIGPIPE 54
 SIGQUIT 54
 SIGSEGV 54
 SIGSTOP 54
 SIGTERM 54
 SIGTSTP 54
 SIGTTIN 54
 SIGTTOU 54
 SIGUSR1 54
 SIGUSR2 54
 sin 25
 sleep 51
 slice 42
 SOCK_DGRAM 48
 SOCK_RAW 48
 SOCK_STREAM 48
 sockaddr_addr 48
 sockaddr_port 48
 socket 48
 sort 43, 44
 space 36
 space? 37
 split 38
 srand 54

- stderr 46
 - stdin 46
 - stdout 46
 - strerror 53
 - strformat (Module) 39
 - string_to_real 46
 - Strings 19
 - strings (Module) 37
 - subscript_ex 30
 - subscript_exception 55
 - substring 37
 - subsystem internal 54
 - subsystem sys 46
 - succ 34, 35, 37
 - sys.dirs (Module) 47
 - sys.errno (Module) 53
 - sys.files (Module) 46
 - sys.net (Module) 48
 - sys.procs (Module) 51
 - sys.signal (Module) 53
 - sys.times (Module) 49
 - sys.users (Module) 49
- T**
- tab 36
 - tail 40
 - take 41
 - tan 25
 - third 45
 - time 49
 - to_int 34, 35
 - to_real 34, 35
 - to_string 30, 33, 34, 35, 36, 37
 - total_gc_time 55
 - tree 29
 - trees (Module) 28
 - triple 45
 - triples (Module) 45
 - Tuple data type 21
- tuple modules 45
 - tuple utilities 45
- U**
- unget 33
 - union (Module) 39
 - unlink 47
 - unpair 45
 - untriple 45
 - unzip 42
 - upcase 37, 38
 - uppercase? 37
 - User-defined data types 21
- V**
- version 54
 - vtab 36
- W**
- wait 51
 - waitpid 52
 - WEXITSTATUS 52
 - whitespace? 37
 - WIFEXITED 52
 - WIFSIGNALED 52
 - WIFSTOPPED 52
 - WNOHANG 51
 - write 47
 - write_char 46
 - wrong_variant_ex 30
 - wrong_variant_exception 55
 - WSTOPSIG 52
 - WTERMSIG 52
 - WUNTRACED 51
- Z**
- zero? 34, 35
 - zip 42

Table of Contents

1	Introduction	1
1.1	About Turtle	1
1.2	Turtle History	2
1.3	Turtle Future	2
2	Using Turtle	3
2.1	Compiling source programs	3
2.2	Command line options	3
2.3	Handcoding	5
2.3.1	Compiling handcoded modules	5
2.3.2	Implementation include file	5
2.3.3	Implementation macros	5
2.3.4	Mapped functions	6
2.4	Documenting Turtle Modules	7
2.4.1	Preparing Turtle Modules for turtledoc	7
2.4.2	Extracting the Documentation	9
3	Language Reference	10
3.1	Turtle Grammar	10
3.1.1	Notation	10
3.1.2	Lexical Structure	10
3.1.3	Turtle Syntax	12
3.1.3.1	Module Syntax	12
3.1.3.2	Declaration Syntax	12
3.1.3.3	Statement Syntax	13
3.1.3.4	Expression Syntax	14
3.1.3.5	Basic syntax items	14
3.2	Turtle semantics	15
3.2.1	Expression Types	15
3.2.2	List operators	15
3.2.3	Array operators	15
3.2.4	List and array expressions	15
3.2.5	Return statements	16
3.2.6	Overloading	16
3.2.7	Generic Modules	16
3.3	Data Types	17
3.3.1	Basic data types	17
3.3.1.1	Integers	17
3.3.1.2	Longs	17
3.3.1.3	Reals	18
3.3.1.4	Booleans	18
3.3.1.5	Characters	18
3.3.1.6	Strings	19
3.3.2	Compound data types	19
3.3.2.1	Array data type	19
3.3.2.2	List data type	20
3.3.2.3	Function data type	20
3.3.2.4	Tuple data type	21
3.3.3	User-defined data types	21
3.4	Runtime environment	22

4	Constraint Programming	23
4.1	Constraints and the Constraint Store	23
4.2	Constrainable Variables	23
4.3	Constraint Statements	23
4.4	Constraints as Assertions	24
5	Standard library	25
5.1	General modules	25
5.1.1	math module	25
5.1.2	random module	25
5.1.3	compose module	26
5.1.4	identity module	26
5.1.5	compare module	26
5.1.6	option module	26
5.1.7	cmdline module	27
5.1.8	hash module	27
5.1.9	hashtab module	28
5.1.10	trees module	28
5.1.11	bstrees module	29
5.1.12	bintree module	29
5.1.13	binary module	30
5.1.14	exceptions module	30
5.1.15	filenames module	31
5.2	Input and output modules	31
5.2.1	io module	31
5.3	Data type related modules	33
5.3.1	ints module	33
5.3.2	longs module	34
5.3.3	reals module	35
5.3.4	bools module	36
5.3.5	chars module	36
5.3.6	strings module	37
5.3.7	union module	39
5.3.8	strformat module	39
5.4	List utility modules	39
5.4.1	lists module	40
5.4.2	listmap module	41
5.4.3	listfold module	41
5.4.4	listreduce module	42
5.4.5	listzip module	42
5.4.6	listindex module	42
5.4.7	listsearch module	43
5.4.8	listsort module	43
5.5	Array utility modules	43
5.5.1	arrays module	43
5.5.2	arraymap module	44
5.5.3	arraysearch module	44
5.5.4	arraysort module	44
5.6	Tuple utility modules	45
5.6.1	pairs module	45
5.6.2	triples module	45
5.7	Low level modules	45
5.7.1	core module	46
5.8	Subsystem sys	46
5.8.1	sys.files module	46
5.8.2	sys.dirs module	47

5.8.3	sys.net module	48
5.8.4	sys.times module	49
5.8.5	sys.users module	49
5.8.6	sys.procs module	51
5.8.7	sys.errno module	53
5.8.8	sys.signal module	53
5.9	Subsystem internal	54
5.9.1	internal.version module	54
5.9.2	internal.random module	54
5.9.3	internal.stats module	55
5.9.4	internal.gc module	55
5.9.5	internal.ex module	55
5.9.6	internal.timeout module	56
5.9.7	internal.limits module	56
Glossary		57
Index		58