The Programming Language DINO

Vladimir Makarov, vmakarov@gcc.gnu.org. Apr 2, 2016

This document describes the programming language DINO – the current language version.

Contents

1 Introduction

DINO is a high level dynamically-typed scripting language. DINO is designed taking such design principles as simplicity, uniformity, and expressiveness into account. Dino is oriented on the same domain of applications as the famous scripting languages Perl, Python, and Lua. Most programmers know the C programming language. Therefore Dino aims to look like C where it is possible. Dino is an object oriented language with garbage collection and pattern matching. Dino has possibilities of a concurrency execution and exception handling. Dino is an extensible language with the possibility of dynamic compilation and loading of a code written on the C language. The high level structures of Dino are

- heterogenous extensible vectors
- extensible associative tables with the ability to delete table elements

- objects

Originally, Dino was used in the Russian graphics company ANIMATEK for a description of the movement of dinosaurs in a project. It has been considerably redesigned and has been re-implemented with the aid of the COCOM tool set.

This document is not intended for use as a programmer's tutorial. It is a concise description of the language DINO and can be used as a programmer's reference.

2 Syntax

An extended Backus-Naur Formalism (EBNF) is used to describe the syntax of Dino. Alternatives are separated by |. Brackets `[` and `]` denote optionality of the enclosed expression, and braces { and } denote repetition (zero or more times). Parentheses `(` and `)` are used for grouping a EBNF construction containing alternatives inside it as one construction.

Terminal symbols denoting a class of terminals (e.g. identifier) consist of only upper-case letters (e.g. IDENT). The remaining terminal symbols either start with a lower-case letter (e.g. the keyword `else`), or are denoted by ASCII character sequences in double quotes (e.g. `"=="`). Non-terminal symbols start with an upper-case letter and contain at least one lower-case letter (e.g. FormalParameters).

3 Vocabulary and Representation

Wherever it is possible, we use also the EBNF for description of lexical symbols through ASCII set characters. Otherwise, we will use natural language sentences in < and >.

Lexical symbols are identifiers, numbers, character constants, strings, operators, delimiters, and comments. White characters (blanks and line breaks) must not occur within the symbols (except in comments, and as the blanks in strings). The white characters are ignored unless they are essential to separate two consecutive lexical symbols. Upper- and lower-case letters are considered to be distinct.

1. An *identifier* is a sequence of letters and digits starting with a letter. The underline is believed to be a valid letter in an identifier. A single underline is fixed for other usage (see a wildcard in section "Patterns").

```
Ident = Letter {Letter | Digit}

Letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
       | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
       | "u" | "v" | "w" | "x" | "y" | "z"
       | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
       | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
       | "U" | "V" | "W" | "X" | "Y" | "Z"
       | "_"

OctalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

Digit = OctalDigit | "8" | "9"
```

```
HexDigit = Digit | "a" | "A" | "b" | "B" | "c" | "C"
         | "d" | "D" | "e" | "E" | "f" | "F"
```

Examples:

```
line  line2  next_line  NextLine
```

2. *Numbers* are (unsigned) integer or floating point numbers. Numbers start with a digit. Numbers starting with the prefix `0x` or `0X` are hexadecimal integer numbers. Otherwise, integer numbers starting with `0` are octal integer numbers. Octal integer numbers should not contain `8` or `9`. If an integer number has the suffix `l` or `L`, it is a long number (see long values in the section "Types and Values"). Floating point numbers are distinguished from decimal integer numbers by the presence of the decimal point `.` or an exponent in the number representation. You can put `_` in number digit sequences unless it is the first symbol of the sequence. It can be useful for readability of long digit sequences.

```
Number = Integer | Long | FloatingPointNumber

DigitSeq = Digit { Digit | "_" }

HexDigitSeq = HexDigit { HexDigit "_" }

Integer = DigitSeq | "0" ("x" | "X") HexDigitSeq

Long = Integer ("l" | "L")

FloatingPointNumber = DigitSeq "." [ DigitSeq ] [Exponent]
                    | DigitSeq [Exponent]

Exponent = ("e" | "E") [ "+" | "-" ] DigitSeq
```

Examples:

```
10
10L
222_222_222_222_222_222_222_222_222_222_222_222_222_222_222_222l
100.
1e2
1000.000_1E+0
1___000__000_000
0xafad_1f34_17ff_
```

3. A Dino *character constant* denotes an Unicode character. The following sequences starting with the ASCII backslash have a special meaning inside a Dino character constant:

- `\a` - ASCII character alert
- `\b` - ASCII character backspace
- `\f` - ASCII character form feed
- `\n` - ASCII character new line
- `\r` - ASCII character carriage return

- `\t` - ASCII character horizontal tab
- `\v` - ASCII character vertical tab
- `\code` - A character with the code given by up to tree octal digits
- `\xcode` - A character with the code given by two hexdecimal digits
- `\ucode` - A character with the code given by four hexdecimal digits
- `\Ucode` - A character with the code given by eight hexdecimal digits
- `\char` - The character char for all remaining characters

To denote a single quote mark use the sequence `\'`. The double quote mark can be represented either by `\"` or simply by `"`. To represent a backslash inside the character constant, use two consecutive ASCII backslashes.

```
Character = "'" Char "'"

Char = <any character except for the single quote ',
        backslash \, or line break>
     | SimpleEscapeSeq
     | OctalEscapeSeq

SimpleEscapeSeq = <one of  \'  \"  \\  \a  \b  \f  \n  \r  \t  \v>

OctalEscapeSeq = "\" OctalDigit [ OctalDigit [ OctalDigit ] ]
```

Examples:

```
'a'  '\''  '\\'  '\12'  '"'
```

4. A *string* is a sequence of characters enclosed in double quotes. There are the same sequences of characters with special meaning as in a character constant. To denote a double quote mark use the sequence `\"`. The single quote mark can be represented either by `\'` or simply by `'`. To represent a backslash inside the character constant, use two consecutive ASCII backslashes.

```
String = '"' {Char} '"'
```

Examples:

```
"This is Dino"  "Don't worry\n"
```

Another variant of a string representation uses back-quotes `` ` ``. A character inside the back-quotes are present in the string as it is, in other words, the escape sequences do not work in such representation. To denote the back-quote in such string representation use double back-qoutes. The newline may not be in such string representation. It means that a string with back-quotes can reside only on one program line.

```
String = '`' {Char} '`'
```

Examples:

```
`\p{Greek}+`  `back qoute `` is here`
```

5. A *C code* is a sequence of characters enclosed in the special brackets `%{` and `%}`. It represents a C code fragment which is compiled and loaded into Dino interpreter during the program execution

```
C_CODE = "%{" <any char sequence not containing pair %}> "%}"
```

An example:

```
%{ static val_t dino_var; %}
```

6. The remaining essential lexical symbols are called *operators* and *delimiters*. Operators are used for forming expressions, delimiters are used for forming syntax constructions. There is a special kind of operators and delimiters which look like identifiers containing only lower-case letters. They are reserved identifiers (keywords). Keywords can not be used in the place of an identifier.

```
OperatorOrDelimeter = "?" | ":" | "|" | "||" | "&" | "&&" | "^"
                    | "==" | "!=" | "===" | "!==" | "<" | ">"
                    | "<=" | ">=" | "<<" | ">>" | ">>>" | "@"
                    | "+" | "-" | "/" | "*" | "%" | "!" | "~"
                    | "#" | ".+" | ".*" | ".&" | ".^" | ".|"
                    | "(" | ")" | "[" | "]" | "{" | "}"
                    | "." | "," | ";" | "=" | "*=" | "/="
                    | "%=" | "+=" | "-=" | "@=" | "<<=" | ">>="
                    | ">>>=" | "&=" | "^=" | "|=" | "++" | "--"
                    | "..." | Keyword

Keyword = "_" | "break" | "case" | "catch" | "char" | "class"
        | "continue" | "else" | "expose" | "extern" | "final"
        | "fiber" | "float" | "for" | "former" | "friend" | "fun"
        | "hide" | "hideblock" | "if" | "in" | "int"
        | "later" | "long" | "new" | "nil" | "obj"
        | "pmatch" | "priv" | "pub" | "return" | "rmatch"
        | "tab" | "thread" | "this" | "throw" | "try" | "type"
        | "use" | "val" | "var" | "vec" | "wait"
```

7. *Comments* are considered analogous to blanks on the syntax level of the program. There are two types of the comments. The first type is an arbitrary character sequence starting with `/*` and finishing with `*/`. The second type comment starts with `//` and finishes with the first line break or with the end of file.

```
Comment = "/*" <arbitrary char. sequence not containing pair */> "*/"
        | "//" <arbitrary char. sequence finishing on line break>
```

4 Declarations and Scope Rules

A Dino program is block structured. Each block introduces a new identifier scope. A block consists of executive statements and declarations and may contain nested blocks. There are also implicit blocks containing each *case*-part of match statements (see below). Each identifier used in a program should be declared in a declaration in the program, unless it is a predeclared identifier.

```
Block = "{"  StmtList "}"

StmtList = { Stmt }

Stmt = ExecutiveStmt
```

```
                    | Declaration
```

When declaring an identifier, you also specify certain permanent properties of a declaration, such as whether it is a variable, a function, a class, or a singleton object. The identifier is then used to refer to the associated declaration (more correctly to the declaration instance).

```
Declaration = VarDeclarations
            | AccessClause
            | ExternDeclarations
            | FuncClassDeclaration
            | SingletonObject
            | ForwardDeclaration
            | IncludeDeclaration
```

The scope of a declaration is textually from the point of the declaration to the end of the block to which the declaration belongs and hence to which the declaration is local. The declaration scope can stop earlier at the point of another declaration with the same identifier in the same block. The declaration scope excludes the scopes of declarations with the same identifier which are in nested blocks.

It is important to understand the notion of *instantiation* of the declaration. This notion reflects a program execution, not the static structure of the program. An instance exists in a *context*. Actually, a context is an execution environment consisting of the covering block instances and/or class objects. A new instance of the block is created when an execution of the block starts. There may be more than one instance of the same block, e.g. when the block is a function or class body (in this case the block instance is a class object), or when the block is executed on different threads (concurrently executed parts of the program) or when there is a reference to a block instance after its execution. When a new instance of the block starts, all the block declarations are instantiated too. For a variable declaration, it means a new instance of the variable is created in the given context. For a function or class declaration, it means that the function or class is bound to the given context.

Example: The following program illustrates a case when a reference to a block instance exists after its execution. The program outputs the result 8.

```
var i, f;

for (i = 0; i < 10; i++)
  if (i % 4 == 0)
    {
      var j = i;
      fun r () {return j;}
      f = r;
    }
putln (f ());
```

There are certain rules for declaration accessibility. Declaration is always either *private* or *public*. Private declaration is accessible only inside the declaration scope or inside functions or classes which are declared as a *friend* in the declaration block. A public declaration instance is always accessible when association (see below) of the identifier is successful. By default, (instances of) declarations in a class block are public. In all other places, the (instances of) declarations are private by default. The following constructions are used for declaring an identifier to be a friend:

```
FriendClause = friend IDENT { "," IDENT } ";"
```

Examples:

```
friend class2;
```

Association of an identifier and the corresponding declaration instance is performed by the following rules:

- The corresponding declaration instance is searched for a separate identifier occurrence in the instance of the block in which the identifier occurs. The identifier should be in the scope of the corresponding declaration. If there is no such declaration, the declaration is searched in the covering block instance of the current block instance back from the current block, and so on. In any case, the identifier should be in the scope of the corresponding declaration.
- Declaration instance for an identifier in the following construction

  ```
  designator.identifier
  ```

  is searched in the block instance (e.g. in a class object) whose value is in the designator. The scope of the found declaration should always include the block end. The exception `accessop` occurs if the declaration is not found with such identifier, or the declaration is private and the construction is not inside a friend of the declaration scope.

The following identifiers are predeclared on the top level (in an implicit block covering the whole program). They are described in more detail later in the report.

```
lang        io          sys         math
re          yaep
```

4.1 Variable Declarations

Dino is an imperative language. In other words it has *variables* which are named containers of values. A variable can contain any value. This means that DINO is a dynamically-typed language. The declaration of a variable also may define the initial value of the variable. In this case a *pattern* can stand on the left side of the assignment instead of just an identifier. The pattern should contain at least one variable. The pattern variables are declared at the assignment point. The pattern should match the assigned value, otherwise the exception `patternmatch` is generated. More details about the patterns, the pattern variables, and pattern matching are decribed later in this document.

Assigning of the initial value to the variable instance is made after execution of the previous statements of the block and after execution of previous assignments in the variable list. By default the initial value of variables is undefined. Nothing can be done with this value. This value even can not be assigned.

The value of the variable is final and can not be changed after its initialization if its declaration is in a list starting with keyword `val`.

Keywords `pub` and `priv` can be used to redefine the default accessibility. They make the declared variables correspondingly public and private.

```
VarDeclarations = [pub | priv] (val | var) VarList ";"
```

```
VarList = Var { "," Var }

Var = IDENT | Pattern "="  Expr
```

Examples:

```
var i = 0, [el1, el2] = [2, 5], j, k;
val constant = 10, nil_constant = nil;
```

4.2 External Declarations

Dino permits to use functions written on the programming language C. The functions should have special prototypes and can use the DINO standard procedural interface (SPI). Dino can also have an access to variables of a special type declared in the C source code. The details of the implementation of such features and the DINO SPI are not described here (some details are given in appendix B). As a rule, the external functions and variables will be implemented as dynamically loaded libraries. This is a powerful instrument of DINO extension. The external functions and variables are declared after keyword `extern`. An external function identifier is followed by `()`. All external declarations (e.g. in different blocks) with the same identifier refer the the same external function or variable. As in the variable declaration, keywords `pub` and `priv` can be used to redefine the default accessibility of the external functions and variables..

```
ExternDeclarations = [pub | priv] extern ExternItems ";"

ExternItems = ExternItem { "," ExternItem }

ExternItem = IDENT
           | IDENT  "(" ")"
```

Examples:

```
extern function (), variable;
```

4.3 Functions and Classes

A function/class declaration consists of a function/class *header*, *formal parameters*, and a function/class block (*body*). The header specifies the function/class identifier.

A function can return the result with the aid of the statement *return*. If the result value after the keyword *return* is absent or the return statement is not executed, the function result is undefined vlaue. An expression-statement as the textually last statement in the function block is actually abbreviation of a return with the expression.

A class call returns the class block instance (an *object* of the class). Any return-statement in a class must be without a result.

Fiber-functions (we call them also simply *fibers* for brevity) are analogous to general functions. The difference is in that a new execution *thread* is created during the fiber call, the return-statement inside the fiber must be without an expression, and the fiber returns the corresponding thread. The thread finishes when the corresponding fiber block finishes. Threads are executed concurrently. Originally only one thread (called the *main thread*) exists in a DINO program.

The formal parameters are considered to be declared in a function/class block and to be initialized by values of the corrsponding *actual parameters* during a call of the function/class. By default they can change their value. Keyword `val` prevents changing value of the corresponding formal parameter after the initialization.

Default accessibility of the formal parameter can be changed by using keywords `pub` or `priv` (see accessibility in the section "Declarations and Scope Rules").

The number of *actual parameters* should be the same as the number of formal parameters. There are two exclusions to this requirement. One exclusion is formal parameters initialized by a default value in a way analogous to the variable initialization. The actual parameters corresponding to the formal parameters with default values can be omitted. In this case, the formal parameters will have the default value. Another exclusion is an usage of `...` at the end of the list of formal parameter declarations, the number of actual parameters can be more the number of formal parameters. Using `...` results in that the formal parameter with the identifier `args` will be declared implicitly. The value of the parameter will be a vector whose elements will be the remaining actual parameter values. If the number of actual parameters is equal to the number of formal parameters (not taking the implicit parameter `args` into account), the value of `args` will be the empty vector.

All formal parameters with default values should be at the end of the parameter list. Usage of formal parameters with default values is prohibited in a case when `...` is used.

If a class contains a function with the name `destroy`, the function will be called when the class object becomes garbage during the garbage collection process or at the end of the program. The function can also be called explicitly from outside if it is declared as public. You should remember that the function is called by the garbage collector without actual parameters and the garbage collector (or finishing the program) ignores the result value. So the function should satisfy the above rules to be called without actual parameters.

You may prevent removing the corresponding object by the garbage collector in the function destroy by assigning the object to a variable. It means that the function can be called several times (during several garbage collections) for the same object. But you should also avoid creation of objects during the call of function `destroy` because it may result in uncontrolled increase of the heap.

A function/class declaration can have optional qualifiers. The qualifier `final` can not be mentioned in an use-clause (see the section "Use-Clause"). Qualifiers `pub` and `priv` change the default accessibility of declarations.

```
FuncClassDeclaration = Header FormalParameters Hint Block

Header = [Qualifiers] FuncFiberClass IDENT

Qualifiers = pub | priv | final
           | pub final | priv final
           | final pub | final priv

FuncFiberClass = fun
               | fiber
               | class

FormalParameters =
                 | "(" [ ParList ] ")"
                 | "(" ParList "," "..." ")"
                 | "(" "..." ")"
```

```
ParList = Par { "," Par}

Par = [pub | priv] [val | var] IDENT [ "=" Expr]

Hint = [ "!" IDENT ]
```

Examples:

The following is parameterless class:

```
class stack () {}
class stack {}
```

The following is a class header with initialization parameters:

```
class stack (max_height = variable,
             val a = 1, priv val b = 2)
```

The following is a function with a variable number of parameters:

```
fun print_args (...) {
    for (i = 0; i < #args; i++)
      println (args[i]);
}
```

The following example is a class with the function `destroy`:

```
var objs_number = 0;
class object () {
  priv var n = objs_number;
  objs_number++;
  priv fun destroy () {objs_number--;}
}
```

The following example illustrates the threads:

```
class buffer (length = 3) {
  var b = [length:nil], first = 0, free = 0, empty = 1;
  priv b, first, free, length;
  fun consume () {
    var res;

    wait (!empty);
    res = b [first];
    first = (first + 1) % length;
    wait (1) empty = first == free;
    return res;
  }
  fun produce (value) {
    wait (empty || free != first);
```

```
          b [free] = value;
          free = (free + 1) % length;
          wait (1) empty = 0;
        }
      }

      fiber consumer (buffer) {
          fun produce (value) {
            buffer.produce (value);
            put ("produce: ");
            println (value);
          }
          produce (10);
          produce (10.5);
          produce ("string");
          produce ('c');
          produce (nil);
      }

      fiber producer (buffer) {
        var value;

        for (;;) {
          value = buffer.consume ();
          if (value == nil)
            break;
          put ("consume: ");
          println (value);
        }
      }

      var queue = buffer ();
      consumer (queue);
      producer (queue);
```

According to Dino scope rules, a declaration should be present before any usage of the declared identifier. Sometimes we need to use function/class identifier before its declaration, e.g. in the case of mutually recursive functions. To do this, Dino has forward declarations.

A forward declaration is just a function/class header followed by semicolon:

```
        ForwardDeclaration = Header ";"
```

A forward declaration denotes the first declaration with the same identifier in the same block after the forward declaration. The forward and the corresponding declaration should be the same declaration type (function, class, or fiber), and have the same accessibility and the same qualifier `final`.

The corresponding declaration of a forward declaration may be absent. In this case a call of the function/class results in occuring the exception `abstrcall`. When accessing to such function through the corresponding object, exception `accessvalue` is generated. A forward declaration without the corresponding declaration can be usefull to describe *abstract classes*. The following are examples of forward declarations:

```
fun even;
fun odd (i) { if (i == 0) return 0; return even (i - 1);}
fun even (i) { if (i == 0) return 1; return odd (i - 1);}
class operation { // abstract class
  fun print_op (); fun apply ();
}
```

A function/class declaration may have an optimization hint. Currently, there are only 3 hints. Hint `!pure` means that the function is pure, i.e. it has no side effects and returns a result depending only on the call arguments. Hint `!inline` means inlining all the function calls. Hint `!jit` means a usage of a JIT compiler for the function/class execution.

4.4 Use-Clause

Dino has a powerful block composition operator *use*. Using it inside a class block can emulate *(multiple) inheritance, traits, duck typing, and dynamic dispatching*. The use-clause has the following syntax:

```
UseClause = use IDENT { UseItemClause }

UseItemClause = [former | later] UseItem { "," UseItem }

Item = IDENT [ "(" IDENT ")"]
```

Use-clause provides a safe way to support object oriented programming. It has the following semantics:

- Declarations of class/function with the identifier given after the keyword *use* are inlayed.
- Declarations before the use-clause rewrite the corresponding inlayed declarations mentioned in *former*-items.
- Declarations after the use-clause rewrite the corresponding inserted declarations mentioned in *later*-items.
- The original (rewritten) and new (rewritting) declarations should be *present* if they are given in former- or later-items.
- The original and new declarations should be *matched*. It means that they should be the same declaration type (variable, function, class, fiber, and external), they should have the same accessibility, they should have the same final attribute for functions, classes, and fibers.
- A rewritten original declaration can be still used in the block if it is *renamed*. The declaration is renamed when a identfier in parentheses is given. The original declaration can be used with this identfier.

The above rules are necessary for correct *duck-typing* or class *sub-typing*.

As a class containing an use-clause provides the same interface as the class mentioned in the use-clause, we also call the using class an *immediate sub-class* of the used class and the used class as an *immediate super-class* of the using class. A class can be used through a chain of several use-clauses. In this more general case we simply use notions of *sub-class* and *super-class*.

The following example illustrates use-clause:

```
class point (x, y) {
}
class circle (x, y, radius) {
  use point former x, y;
  fun square () {3.14 * radius * radius;}
}
class ellipse (x, y, radius, width) {
  use circle former x, y, radius later square;
  fun square () {
    ...
  }
}
```

4.5 Singleton Object

Sometimes it is usefull to guarantee that there is only single object (called *singleton object*) of a class. Although it can be achieved by creating of an object of an anonymous class (see the section "Anonymous Functions and Classes") and assigning it to a variable declared as `val`, Dino has a special declaration for singleton object:

```
SingletonObject = [pub | priv] obj IDENT block
```

The keywords `pub` and `priv` can be used to redefine the default accessibility. They make the declared object correspondingly public and private. Example of singleton object declaration:

```
priv obj coord {
  val x = 0, y = 10;
}
```

You can not refers to a singleton object by its name inside the singleton object as the declaration actually occurs after the object block.

4.6 Expose-clause

Signleton object declarations can be accessed through designator with `.` – see *Designators*. Sometimes it can be too verbose. The expose-clause can make access to public object declarations more concise by introducing declarations on the clause place which are actually aliases to the corresponding object declarations.

```
ExposeClause = expose ExposeItem { "," ExposeItem }

ExposeItem = QualIdent ["(" IDENT ")"] | QaulIdent ".*"

QualIdent = IDENT {"." IDENT}
```

You can expose one declaration by using QualIdent with an optional alias identfier in parentheses. If the alias is not given, the declaration identifier will be used for the alias. The qualident designates the declaration which is inside one signleton object or inside the nested singleton objects.

You can also expose all public declarations of a signleton object by their identifiers. In this case, you need to add `.*` after the qualident designating the corresponding object.

It is a good practise to put all package declarations in a singleton object which in this case behaves as *name space* or modules in other programming langiages.

Example of use-clauses:

```
expose sys.system (shell), math.*;
shell ("echo sin:" @ sin (3.14) @ "cos: " @ cos (3.14));
```

In the above example, we make the function `system` from singleton object `sys` accessible through identifier `shell` and all public declarations from singleton object `math` including `sin` and `cos` accessible by their identifiers.

5 Expressions

Expressions are constructs denoting rules of computation of a value from other values by the application of *operators*. Expressions consist of *operands* and operators. Parentheses may be used to express specific associations of operators and operands.

5.1 Types and Values

All Dino values are *first class values*, i.e. they can be assigned to a variable, can be passed as a parameter of a function/class, and can be returned by functions. Operators require operands whose values are of given type and return the value of the result type. Most values have a representation in Dino. When a value representation is encountered in an expression during the expression evaluation, the new value is generated.

There are values of *structured types*, i.e. values which are built from other values. The value of a structured type may be *mutable* or *immutable*. A value or sub-value of a mutable value can be changed. An immutable value can not be changed after its generation. You can make a mutable value immutable as a side effect by applying the operator `final` (the table key is also made immutable as a side effect of writing to the table). In all cases, the operator returns the operand value as the result. If you try to change an immutable value, the exception `immutable` is generated. You can make a new mutable value as a side effect of applying operator `new`. The operator returns a new value equal to the operand value.

```
Expr = final  Expr
     | new  Expr
```

Structured value types are also *shared value types*. This notion means that if two or more different variables (array elements or table elements) refer to the same value and the value is changed through one variable, the value which is referred through the other variables is changed too. There is no difference between the notion "the same value" and the notion "equal values" for non-shared type values. For the shared type operands, equality means that the operands have the same structure (e.g. vectors with the same length) and the corresponding element values are the same.

Examples:

```
new 5
new ['a', 'b', 'c']
new "abc"
new tab ["key0" : 10, "key1" : 20]
final 5
final ['a', 'b', 'c']
final "abc"
final tab["key0" : 10, "key1" : 20]
```

Dino has the following types of values:

- the special value *nil*. The value is represented by the keyword `nil`.

  ```
  Expr = nil
  ```

- A *character* which represents unicode characters. For the representation see `Character` in the section *Vocabulary and Representation*.

  ```
  Expr = CHARACTER
  ```

- An *integer*. For its representation see `Integer` in the section *Vocabulary and Representation*. It is always stored as a 64-bit integer value.

  ```
  Expr = INTEGER
  ```

- A *long* integer. For its representation see `Long` in the section *Vocabulary and Representation*. Long values are signed multi-precision integer. The maximal and minimimal values are constrained only by the overall computer memory size.

  ```
  Expr = LONG
  ```

- A *floating point number*. For its representation see `FloatingPointNumber` in the section *Vocabulary and Representation*. It is always stored as an IEEE double (64-bit) floating point value.

  ```
  Expr = FLOATINGPOINTNUMBER
  ```

- A *vector*. This is a structured shared type value. A vector value is represented by a list of values (or expressions) in brackets with optional repetitions of the vector elements preceded by `:`. The repetition value is converted into an integer value by default. If the repetition value after the conversion is not integer, the exception `optype` is generated. If the repetition value is negative or zero, the element value will be absent in the vector. Elements of a vector are accessed by their indexes. Indexes always starts with 0. Vectors in Dino are *heterogenous*, i.e. elements of a vector may be of different types. A string represents an immutable vector all of whose elements are characters in the string. Elements of mutable vectors can be added to or removed from the vector (see predefined functions *ins, insv, and del*).

  ```
  Expr = "["  ElistPartsList "]"
       | STRING
  ElistPartsList = [ Expr [":" Expr ] {"," Expr [":" Expr ] } ]
  ```

  Examples:

  ```
  "aaab"
  ['a', 'a', 'a', 'b']
  [3 : 'a', 'b']
  ```

```
[3.0 : 'a', 'b']
["3" : 'a', 'b']
['a', 10, 10.0, "abcd", {}]
[]
```

- A *table*. This is a structured shared type value. A table value is represented by keyword tab, followed by a list of key values (expression values) in brackets `[` and `]` with optional element values with a preceding `:`. By default the element value is equal to `nil`. It is not allowed to have elements with equal keys in a table. If this constraint is violated in a table constructor, the exception `keyvalue` is generated. Elements of tables are accessed by their keys. Elements of mutable tables can be added to or removed from the table correspondingly by assigning values and with the aid of the function *del*. The side effect of the table constructor execution is that the keys become *immutable*.

  ```
  Expr = tab "["  ElistPartsList "]"
  ```

  Examples:

  ```
  tab ['a', 'b', 10:[10]]
  tab ['a' : nil, 'b' : nil, 10 : [10]]
  tab [[10, 'a', tab [10]] : 10, [10] : tab [20:20]]
  tab []
  ```

- A *function*. Its value is represented by the function designator. It is important to remember that the function is bound to a context.
- A *fiber-function*. Its value is represented by the fiber-function designator. It is important to remember that the fiber is bound to a context.
- A *class*. Its value is represented by the class designator. It is important to remember that the class is bound to a context.
- A *thread*. There is no literal Dino representation of such values. A thread is generated by calling a fiber.
- An *object* (a block instance, usually a class block instance). This is a structured shared type value. There is no literal Dino representation of such values. Objects are generated by the class block instantiation and can be accessed by value `this` immediately inside the class.
- A *hide value*. A hide value can not be generated by a Dino code. They are generated by external functions.
- A *hide block*. This value is analogous to a hide value. The differences are in that the size of a hide value is constrained by a C program pointer. The size of a hideblock value has no such constraint. Also a hideblock is of shared type.
- A *type*. The values of such types are returned by the special operator `type (expression)`.

  ```
  Expr = char
       | int
       | long
       | float
       | hide
       | hideblock
       | vec
  ```

```
                                     | tab
                                     | fun
                                     | fiber
                                     | class
                                     | thread
                                     | obj
                                     | type
```

There are the following type values:

– The type of `nil`. There is no value representing type of `nil`. So use the construction `type (nil)` to get it.
– The type of characters. The value is represented by the Dino keyword `char`.
– The type of integers. The value is represented by the Dino keyword `int`.
– The type of long integers. The value is represented by the Dino keyword `long`.
– The type of floating point numbers. The value is represented by the Dino keyword `float`.
– The type of vectors. The value is represented by the Dino keyword `vec`.
– The type of tables. The value is represented by the Dino keyword `tab`.
– The type of functions. The value is represented by the Dino keyword `fun`.
– The type of fiber-functions. The value is represented by the Dino keyword `fiber`.
– The type of classes. The value is represented by the Dino keyword `class`.
– The type of block instances (objects). The value is represented by the Dino keyword `obj`.
– The type of threads. The value is represented by the Dino keyword `thread`.
– The type of hide values. The value is represented by the Dino keyword `hide`.
– The type of hideblocks. The value is represented by the Dino keyword `hideblock`.
– The type of types. The value is represented by the Dino keyword `type`.

5.2 Designators

There is a special Dino construction called a *designator*. A designator refers for a vector or table element or for a declaration. If the designator refers to a vector or table element or for a variable declaration, it can stand in the left hand side of an assignment statement. If the designator stands in an expression, the corresponding value is used (the vector/table element value, variable value, function, fiber, or class). When the designator referring to a table element stands up in the left hand side of an assignment statement, *its key value becomes immutable*.

A designator referring to a vector or table element has the following syntax:

```
                Designator = (Designator | Call) "["  Expr "]"
```

The value of the construction before the brackets must be a vector or table. Otherwise, the exception `indexop` is generated.

If the value of the construction before the brackets is a vector, the value of expression in the brackets (so called *index*) is converted to an integer value. If this is not possible, the exception `indextype` is generated.

If the index is negative or greater than or equal to the vector length, the exception `indexvalue` is generated. The value of the designator will be the vector element value with given index (the indexes starts with zero). Examples:

```
vector [1]
vector ["1"]
vector [1.0]
```

If the value of the construction before the brackets is a table, the value of expression in the brackets is called *key*. The value of the designator will be the table element value with the key which is equal to given key. If the element with the given key is absent in the table, the exception `keyvalue` is generated. Examples:

```
table ['c']
table [10]
table ["1"]
table [1.0]
```

The remaining forms of designator refer to a declaration. See the section *Declarations and Scope Rules* for a description on how they work.

```
Designator = (Designator | Call) "."  IDENT
           | IDENT
```

Examples:

```
value
value.f
```

Dino also has an extended form of the designator described above. This form is called a *vector slice*:

```
Expr = ExtDesignator

ExtDesignator = Designator {Slice}

Slice = "["  [Expr] ":" [Expr] [":" Expr] "]"
```

The slice is applied to a vector, otherwise the exception `sliceform` is generated. The slice refers for a part of vector given by the start index, the bound, and the step. If they are not integer, the exception `slicetype` is generated.

The negative bound means counting the index from the vector end. `-1` corresponds to the vector length, `-2` corresponds to the vector length `-` 1, `-3` corresponds to the vector length `-`2, and so on. If the bound is more than the vector length, the bound is assumed to be equal to the vector length. If the bound is omitted, it is believed to be equal `-1`, or in other words, to the vector length.

If the start index is less than zero, the exception `sliceform` is generated. If the start index is omitted, it is believed to be zero. If the step is omitted, it is believed to be one.

The slice refers to vector elements with indexes `start`, `start + abs (step)`, `start + 2 * abs (step)`, `start + 3 * abs (step)` and so on while the index is less than the bound. If the step value is negative, the elements are considered in the reverse order. Examples:

```
v[:]    // all vector elements
v[1:]   // all vector elements except the first one
v[:-2]  // all vector elements except the last one
v[::-1] // all vector elements in reverse order
v[0::2] // all vector elements with even indexes
```

More one slice can be given after the designator. In this case, next slice is applicable to each element of the previous slice. That means that each element and the next slice should abide by the above rules. The first slice is called the first *dimension* slice, the second one is called the second dimension slice and so on. Examples of multi-dimensional slices:

```
m[:][:]     // all matrix elements
m[:][::2]   // matrix elements with even columns
m[::2][::]  // matrix elements with even rows
m[::2][::]  // matrix elements with even rows and columns
```

The slices are actually not real reference values. They can be considered as an attribute which exists in a statement at most. If vector with slices is an operand of an operation which has a slice variant, the operation is applied to each element of the vector referenced by the slices in given order. The result is a new vector with the same dimension slices referencing all elements with the step value equal to one.

When a vector with slices is an operand of an operation, the new vector consisting of elements referenced by the slices is created. The new vector will have the same dimension slices referencing for all new vector elements with the step equal one. This is done to avoid unexpected side-effects of function-calls inside a statement with slices.

5.3 Calls

One form of expression is a call of a function, a fiber, or a class. The value of the designator before the actual parameters should be a function, a fiber, or a class. Otherwise, the exception `callop` is generated. An instance of the block corresponding to the body of the function, fiber, or class is created. The actual parameter values are assigned to the corresponding formal parameters. If the corresponding function, fiber, or class has no default formal parameter `args` or parameters with default values (see the section *Declarations*), the number of the actual parameters should be equal to the formal parameter number. Otherwise, a vector whose elements are the remaining parameter values is created and assigned to the parameter `args`. If there is no corresponding actual parameter for a formal parameter with a default value, the default parameter value (see the section *Declarations*) is assigned to the formal parameter. After the parameter initialization the statements in the block are executed. If it is the call of a fiber, a new execution thread is created, and the statements of the block is executed in the new thread. The value of call of the fiber is the corresponding thread. It is returned before starting the execution of statements in the new thread.

The execution of the body is finished by reaching the block end or by execution of a return-statement. Finishing of the fiber results in finishing the corresponding thread. The return-statement in a fiber or in class should be without an expression. The call of a class returns the created object. A function call returns the value of the expression in the executed return-statement. Otherwise, the function return value is undefined.

```
Expr = Call
```

```
Call = Designator ActualParameters

ActualParameters = "(" [ Expr { "," Expr } ] ")"
```

Examples:

```
f ()
f (10, 11, ni, [])
obj.objf ()
```

5.4 Operators

Expressions consist of operands and operators. The order in which operators are executed in an expression is defined by their *priority* and *associativity*. That means that the expression `a op1 b op2 c` when the operator `op2` has higher priority than `op1` is analogous to `a op1 (b op2 c)`. Dino operators have analogous priorities to the ones in the language C. The following Dino operators are placed in the order of their priority (the higher the line on which the operator is placed, the higher its priority).

```
!  #  ~  final  new
*  /  %
+  -
@
<<  >>  >>>
<  >  <=  >=
==  !=  ===  !==
&
^
|
in
&&
||
:
?
```

All binary operators have left associativity in Dino. That means that the expression `a op1 b op2 c` when operators `op1` and `op2` have the same priority is analogous to `(a op1 b) op2 c`. Parentheses may be used to express specific associations of operators and operands.

```
Expr = "(" Expr ")"
```

Most of the Dino operators require the operands to be of given types. If an operand is not of given type, the conversion of it into a value of the type needed may be made. If after the possible conversions the operands are still not of necessary types, the exception `optype` is generated. The following conversions may be made by default:

- An *integer conversion*. If the operand is a character, its code becomes integer. If the operand is a long integer, it becomes integer. If the long integer requires more 64 bits, the exception `opvalue` is generated. If the operand is a floating point number, its fractional part is thrown away and integral part becomes integer. If the operand is a vector of characters, the corresponding string is believed to be the decimal representation of an integer and is converted into the corresponding integer value. If the corresponding string is not a correct integer representation, the result is undefined. If the corresponding string represents an integer whose representation requires more 64 bits, the exception `erange` may be generated. In all remaining cases the results of conversion coincide with the operand.

- An *arithmetic conversion*. Analogous to integer conversion except for that the conversion of a long integer or a float pointing number to integer is not made and if the string represents a long integer (i.e. contains suffix `l` or `L`) or a floating point number (i.e. contains an exponent or fraction), the result will be the corresponding long integer or floating point number instead of an integer. If the result of conversion to an integer from the string is out of range of 64-bit representation or the result of conversion to floating-point number from the string is out of IEEE double format range, the exception `erange` may be generated. Additionally if the operand is in a non-short circuit binary operator (non-logical operators) and another operand is a floating point number after the conversion, the first operand is converted into a floating point number too. In this case, if the first operand is a long integer out of IEEE double format range, the result is undefined. Otherwise, if another operand is a long integer after the conversion, the first operand is converted into long integer too.
- *String conversion*. If the operand is a character, the result will be a new string (immutable vector of characters) with one element which is the character. If the operand is an integer, a long integer, or a floating point number, the result will be a new string of characters which is a decimal string representation of the number (long will have no suffix).

5.4.1 Logical operators

Logical operators produce the integer result 1 which means *true* or 0 which means *false*. Logical 'or' || and logical 'and' `&&` are *short circuit* operators. That means that the second operand is evaluated depending on the result of the first operand. When the operands of the operators are evaluated, the arithmetic conversion is made.

If the first operand of logical 'or' is nonzero (integer, long integer, or floating point), the result will be 1. Otherwise, the second operand is evaluated. If the second operand is nonzero, the result will be 1. Otherwise, the result will be 0.

If the first operand of logical 'and' is zero (integer, long integer, or floating point), the result will be 0. Otherwise, the second operand is evaluated. If the second operand is nonzero, the result will be 1. Otherwise, the result will be 0.

Logical negation `!` makes an implicit arithmetic conversion of the operand. If the operand is zero (integer, long integer or floating point), the result will be 1. Otherwise, the result will be 0.

Operator `in` checks that there is an element with the given key (the first operand) in the given table (the second operand). If the element is in the table, the result will be 1. Otherwise the result will be 0. If the second operand is not a table, the exception `keyop` is generated.

```
Expr = Expr "||"  Expr
     | Expr "&&"  Expr
     | Expr in  Expr
     | "!"  Expr
```

Examples:

```
!(type (i) == int && type (a) == tab && i >= 0 && i < #a)
k in t && t[k] == 0
0.0  || another_try
0  || another_try
```

5.4.2 Bit operators

The following operators work on integers (implicit integer conversion is made) and return an integer result. Operators | `^ & ~` denote correspondingly *bitwise or, bitwise exclusive or, bitwise and*, and *bitwise negation* of 64-bit integers.

Operators << >>> >> denote correspondingly *logical left bit shift, logical right bit shift*, and *arithmetic right bit shift* (with sign extension) of given number (the first operand) by given number of the bits (the second operand). The value of the second operand must be non-negative, otherwise the result is undefined.

```
Expr = Expr "|"  Expr
     | Expr "^"  Expr
     | Expr "&"  Expr
     | Expr "<<"  Expr
     | Expr ">>"  Expr
     | Expr ">>>"  Expr
     | "~"  Expr
```

Examples:

```
(i >> shift) & mask
i & ~mask | (value << shift) & mask
i >>> 2
i << 2
```

5.4.3 Comparison operators

All comparison operators return a logical value (integer 0 which means false or integer 1 which means true).

Operators equality `==` and inequality `!=` may make some conversion of the operands. If one of the two operands is string, then the string conversion is applied to the other operand before the comparison. Otherwise, standard arithmetic conversion is applied to the operands if one operand is of an arithmetic type. The operators do not generate exceptions (but the conversions may). The operands are equal if they have the same type and equal values (see the section *Types and Values*). For instances, functions and classes, the equality requires also the same context.

Operator identity `===` or unidentity `!==` returns 1 if the operands have (or not) the same value or 0 otherwise. The operators never generate exceptions and no any conversion is made.

By default the arithmetic conversion is applied to the operands of operators < > `<=` `>=`. There is no exception if the operands after the conversion are of integer, long integer, or floating point type. So the operands should be characters, integers, long integers, floating point numbers, or strings representing integers, long integers, or floating point numbers.

```
Expr = Expr "=="  Expr
     | Expr "!="  Expr
     | Expr "==="  Expr
     | Expr "!=="  Expr
     | Expr "<"  Expr
     | Expr ">"  Expr
     | Expr "<="  Expr
     | Expr ">="  Expr
```

Examples:

```
10 == 10
10 === 10
10 == 10l
10 == 10.0
10 !== 10l
10 !== 10.0
10 <= 'c'
p != nil
'c' == "c"
10 < "20.0"
[10, 20] == [10, 20]
[10, 20] !== [10, 20]
```

5.4.4 Arithmetic operators

The following operators return integer, long integer, or floating point numbers. Before the operator execution, an implicit arithmetic conversion is made on the operands. The binary operators `+ - * / %` denote correspondingly integer, long integer, or floating point addition, subtraction, multiplication, division, and evaluation of remainder. Unary operator `-` denotes arithmetic negation. The unary operator `+` is given for symmetry and it returns simply the operand after the conversion. It can be used for conversion of a string into an integer, a long integer, or floating point number.

```
Expr = Expr "+"  Expr
     | Expr "-"  Expr
     | Expr "*"  Expr
     | Expr "/"  Expr
     | Expr "%"  Expr
     | "+"  Expr
     | "-"  Expr
```

Examples:

```
+"0"
+"10l"
+"10."
+"1e1"
-i
(value + m - 1) / m * m
index % bound
```

5.4.5 Miscellaneous operators

The Dino conditional expression is analogous to the language C one. An implicit arithmetic conversion is made for the first expression followed by `?`. If the value of the expression is non zero (integer, long integer, or floating point value), the second expression with following `:` is evaluated and it will be the result of the condition expression. Otherwise, the third expression is evaluated and it becomes the result.

The operator `#` can be applied to a vector or a table. It returns the length of the vector or the number of elements in the table.

The operator `@` denotes a concatenation of two vectors into a new vector. Before the concatenation an implicit string conversion of the operands is made.

The remaining operators look like function calls. Operator `type` returns the expression type. Never is an exception generation possible during the operator evaluation.

The operator `char` is used to conversion of a value into a character. First, an implicit integer conversion is applied to the operand. The operand should be an integer after the conversion. Otherwise, the exception `optype` will be generated. The integer is transformed into the character with the corresponding code. If the code is too big to be a character or is negative, the exception `erange` is generated.

The operator `int` is used for a conversion of a value into an integer. Implicit integer conversion is applied to the operand. The operand should be an integer after the conversion. Otherwise, the exception `optype` will be generated. If the code is too big to be an integer, the exception `erange` is generated.

The operator `float` is used for a conversion of a value into a floating-point number. The first, an implicit arithmetic conversion is applied to the operand. The operand should be an integer, an long integer, or a floating-point number after the conversion. Otherwise, the exception `optype` will be generated. An integer is transformed the corresponding floating-point number. The same is done for a long integer but if the number is too big or too small to be a floating-point number, the result is undefined.

The operator `vec` is used for a conversion of a value into a vector. First, an implicit string conversion is applied to the operand. The optional second expression defines the format used only for the string conversion of a character, an integer, a long integer, a floating point number, or a string. The second parameter value should be a string after an implicit string conversion. The format should not be given for a table. The first operand should be a table or a vector after the conversion. The table is transformed into a new vector which consists of pairs (one pair for each element in the table). The first element of the pair is a key of the corresponding element, and the second one is the element itself. The order of pairs in the result vector is undefined.

The operator `tab` is used for a conversion of a value into table. First, a string conversion is applied to the operand. The operand should be a vector or a table after the conversion. The vector is transformed into a new table whose elements are equal to the vector elements that have integer keys equal to the corresponding vector indexes.

```
Expr = Expr "?"  Expr ":" Expr
     | "#"  Expr
     | Expr "@"  Expr
     | type "(" Expr ")"
     | char "(" Expr ")"
     | int "(" Expr ")"
     | float "(" Expr ")"
     | vec "(" Expr ["," Expr] ")"
```

```
                    | tab "(" Expr ")"
```

Examples:

```
i < 10 ? i : 10
#tab ["a", 'b']
#["a", 'b']
"concat this " @ "and this"
type (type)
type (10)
char (12)
vec  (10)
vec  (10, "%x")
vec (tab ["1":1, "2":2])
tab ([1, 2, 3, 4])
```

5.4.6 Slice Operators

There are slice variants of all operators of mentioned above except for the short circuit operators `&&` and || and the conditional expression.

For unary operators, operator execution on slices creates a new vector structure consisting of elements referenced by the slices in given order, creates the same dimension slices referencing all elements of the new vector structure and execute the operator on each element of the created slices which will be the result of the operator. Examples:

```
var v = [1, 2, "3"], m = [[1, 2.], [3, 4, 5]];
! v[:];
- m[1:][::2];
float (m[:][:]);
```

Binary operators on the slices are a bit more complicated. If the both operands have slices, they should have the same *form* (dimension, number of elements in each corresponding sub-vector referenced by the slices), otherwise the exception `sliceform` is generated. The operator execution can be considered as creating a new vector structure consisting of elements referenced by the slices in and creating the same dimension slices referencing all elements of the new vector structure. The same is done for the second slice. Than the elements of the new vector structure created for the first operand is changed by the result of the operator applied to this element and the corresponding element from the slices created for the second operand. The slices created for the first operand will be the result. Examples:

```
var v = [1, 2, 3], m = [[4, 5], [6, 7]];
v[:] * v[:];
m[:][0::2] + m[:][1::2];
```

If only one operand has the slices, the operator execution can be considered as creating a new vector structure consisting of elements referenced by the slices and creating the same dimension slice referencing all elements of the new vector structure. Than the elements of the new vector structure is changed by the result of the operator applied to this element and another operand in given operand order. The created slices will be the result. Examples:

```
var v = [1, 2, 3], m = [[4, 5], [6, 7]];
1 - v[:];
v[:] * v[:] * 2;
m[:][0::2] + 1;
```

Dino has additional unary operators which work only on the slices:

```
Expr = ".+"  Expr
     | ".*"  Expr
     | ".&"  Expr
     | ".^"  Expr
     | ".|"  Expr
```

The are called *fold operators*. If the operand has no slices, the exception `vecform` is generated. The operator execution is to apply the corresponing non-fold operator to all elements referenced by the slices. Examples:

```
var v = [1, 2, 3], m = [[4, 5], [6, 7]];
.+ v[:];
.* v[:];
.& m[:][:];
```

5.5 Current block instance

The instance of the block immediately surrounding the current program point can be accessed by using the keyword `this`.

```
Expr = this
```

Example of usage of `this`:

```
class c (x) { putln (this.x, "===", x); }
fun f (pub x) { return this; }
```

5.6 Anonymous Functions and Classes

Instead of declaring a function/class and using its identifier once in an expression, an anonymous function/class can be used. The anonymous functions/classes are regular expressions. They have the same syntax as the corresponding declarations. The only difference is an absence of the function/class identifier:

```
Expr = AnonHeader FormalParameters Block

AnonHeader = [Qualifiers] FuncFiberClass
```

Examples:

```
fun (a) {a > 0;}
class (x, y) {}
fiber (n) {for (var i = 0; i < n; i++) putln (i);}
```

An example of anonymous function usage:

```
fold (fun (a, b) {a * b;}, v, 1);
```

5.7 Try-operator

Dino has two ways to process the exceptions. One way is described in the section *Try-block*. Another way is to use a try-operator. The try-operator has the following syntax:

```
Expr = try "(" ExecutiveStmt [ "," ExceptClassList] ")"
```

The operator returns non-zero if no exceptions occurs in the statement given as the first operand. The operator returns zero if an exception occurs and its class is a sub-class (see the function `isa`) of one exception class given by the subsequent operands. If there is no matched operand class, the exception is propagated further (see the *try-block* description for more details). If the optional subsequent operands are omitted, any occured exception in the statement results in the zero result value.

Here is the example of usage of the try-operator:

```
var ln;
for (; try (ln = getln (), eof);) putln (ln);
```

6 Patterns

Patterns have a form of expressions. They are used to match an expression and decompose it. Pattern can have form a vector value, a table value, a function call, an identifier, or a special *wildcard* symbol `"_"`. Anything else in the pattern is treated as an expression. The simplest way is to treat a pattern, e.g. identfier, as an expression is to put it into paretheses.

```
Pattern = Expr
Expr = "_" | "..."
```

A separate wildcard matches any value. Identifier also matches any value but additionally the matched value is assigned to the *pattern variable* denoted by the identifier. An expression in a pattern matches any value equal to the the expression value.

A *vector pattern* matches a *vector value*. All vector pattern elements should match all subsequent elements of the vector value. If a vector pattern element has a form of another pattern than it matches the corresponding vector value element iff the element pattern matches the corresponding element value of the vector value.

A vector pattern element can also have a form `expr ":" pattern`. The value of the expression before `":"` is converted into an integer value by default. If the value after the conversion is not integer, the exception `optype` is generated. The value can be zero or negative. In this case matching the pattern element is always successful and this matching does not correspond any element of the matched vector value. Othewise, the pattern after `":"` should match the corresponding elements in the vector value. The exact number of the elements is defined by the expression value before `":"`. The matched element values should be equal unless the pattern after `":"` is a wildcard `"_"`.

The last element of the vector pattern can be `"..."`. It matches the rest elements in the matched vector value. The following are examples of vector patterns (a, b, and c are identifies of pattern variables):

```
[a, b]   // a and b are pattern variables
[(a), b] // value of earlier declared a is used
[a, ...]
[2, [a, b], 3 : c, 4 : _]
```

A *table pattern* matches a *table value*. All table pattern elements should match all elements of the table value. If a table pattern element has a form `expr`, then the table value should have an element with a key given by the expression.

A table pattern element can also have a form `expr ":" pattern`. In this case the table value also should have an element with a key given by the expression. Additionally the element value with given key should match the pattern after `":"`.

The last element of the table pattern can be `"..."`. It matches the rest elements in the matched table value. The following are examples of table patterns (a, b, and c are identifies of the pattern variables):

```
tab ["k1", "k2"]
tab ["k1" : _, ...]
tab ["k" @ 1, [a, b], "k2" : c, ...]
```

An object pattern has a form of function/class call. An *object pattern* matches a class/function instance. The function/class should be a subtype of class/function given by the value of the expression in the pattern before `"("`. The parameters in the pattern should match all corresponding instance parameter values. The number of the values is taken from function/class given by the expression in the pattern before `"("`, not from function class of the instance. Remember that the arguments corresponding to `"..."` in function/class parameter definitions become one array value.

The last parameter of the object pattern can be `"..."`. It matches the rest arguments of the corresponding object. The following are examples of object patterns (`a, b`, and `c` are identifies of the pattern variables):

```
leaf (10)
node (_, a)
node (node (a, b), leaf (c))
```

Symbols `"_"` and `"..."` can be used only in patterns - they can not be used in expressions.

7 Executive statements

Statements denote actions. There are *simple* and *compound* statements. Simple statements do not consist of any parts that are statements themselves. They are the assignment, procedure call, return, break, continue, throw, and the wait statements. Analogous to the language C the last symbol of a Dino simple statement is semicolon `;`. Compound statements consists of parts that are statements themselves. They are used to express sequencing, exception handling, conditional, and repetitive execution.

7.1 Empty statement

There is an empty statement in Dino. It denotes no action. The empty statement is included in Dino for convenience.

```
ExecutiveStmt = ";"
```

Example: Usage of an empty statement in a for-statement:

```
for (i = 0; a[i] == 0; i++)
  ;
```

7.2 Block-statement

A block-statement is simply a block and can used to group statements into one statement and/or describe local declarations. For details on how the block is executed see the section *Declaration and Scope Rules*.

```
ExecutiveStmt = BlockStmt

BlockStmt = Block
```

Example: Usage of a block-statement in a for-statement:

```
sum = 0;
for (i = 0; i < #a; i++)
  {
    var value = a[i];
    if (value > 0)
      sum += value;
  }
```

7.3 Expression-statement

Dino has an expression statement. Although it seems that the expression value is not used, the expression evaluation can results in side-effects, e.g. through calling a function/class. An expression-statement as textually the last statement in the function block means a return of the expression value as the function call result. In REPL (see Appendix B. Implementation), execution of an expression-statement results in printing the expression value.

```
ExecutiveStmt = Expr ";"
```

Examples:

```
putln ("percent=" @ percent @ "%");
newfiber ();
5 + 10;
```

7.4 Assignment statements

Assignment-statements are used to change variable values or element values of a structured value which are referred through a designator (see sub-section *Designator* in section *Expressions*. The designator can not denote a final variable (see the section *Variable Declaration*). You can not change the element value of an immutable value (see the section *Types and Values*). In this case the exception `immutable` is generated. Assignment to a table element has a side effect, the element key becomes immutable.

A simple assignment statement looks like `Designator = Expr;`. That means that the expression value is assigned to a variable or element of a structured type value denoted by the designator. For the convenience of C programmers there are also the Dino assignments `Designator op= Expr;`, `Designator++;`, `++Designator;`, `Designator-;`, and `-Designator;`. They are analogous correspondingly to `Designator = Designator op Expr;`, `Designator = Designator + 1;`, and `Designator = Designator - 1;`. The only difference is in the fact that the designator is evaluated only once, not twice as in the analogous form. It is important to know if you have *side effects* in the statement.

```
ExecutiveStmt = Designator Assign Expr ";"
              | Designator ("++" | "--")  ";"
              | ("++" | "--")  Designator ";"
Assign = "="
       | "*="
       | "/="
       | "%="
       | "+="
       | "-="
       | "@="
       | "<<="
       | ">>="
       | ">>>="
       | "&="
       | "^="
       | "|="
```

Examples:

```
v = [10, 20];
i = 1;
i++;
--i;
i *= 20;
```

If the designator is a slice and the expression value is a slice too, they both should have the same *form*, otherwise the exception `sliceform` is generated. In this case each element referenced by the designator slice gets value of the corresponding element referenced by the expression slice. If the designator is a slice but the expression value is not, each element referenced by the designator slice gets the expression value. Examples of the slice assignment:

```
v = [1, 2, 3, 4];
v[:] += 1;
v[::-1] = v[:]; // reverse v
```

7.5 If-statement

The Dino if-statement is analogous to the C language one. First, the expression after `if` is evaluated and an arithmetic conversion is done to it. The value should be an integer, a long integer, or a floating-point number, otherwise the exception `optype` is generated. If the value is nonzero the first statement is executed, otherwise the statement after `else` is executed (if any). The problem with *dangling else* is resolved analogous to the language C – `else` part is associated with the closest `if`.

```
ExecutiveStmt = if  "(" Expr ")" Stmt [ else Stmt ]
```

Examples:

```
if (i < 0) i = 0;
if (i < j) return -1; else if (i > 0) return 1; else return 0;
```

7.6 For-statement

The Dino for-statement is analogous to the C language one. The statement is executed in the following way.

1. Execution of the first statement in the parentheses is done.
2. The expression (*for-guard*) is evaluated and an implicit arithmetic conversion is applied to its value. The result value should be an integer, a long integer, or a floating point number. If this is not true, the exception `optype` is generated.
3. If the value of for-guard is nonzero, the body of the loop (the last statement) is executed. Otherwise, the for-statement execution finishes.
4. When the body has been executed, the second statement in the parentheses is executed and steps 2,3,4 (one iteration) are repeated again.

If the second statement is a simple statement, the statement semicolon can be omitted. The for-statement also can be finished by an execution of the statement `break` in the body. The rest body execution can be skipped by an execution of the statement `continue`. In this case, the for-statement execution continues with the step 4.

```
ExecutiveStmt = for  "(" Stmt ForGuardExpr ";"  Stmt ")" Stmt

ForGuardExpr = [Expr]
```

Examples:

```
for (i = 0; i < 10; i++;) sum += v [i];
for (i = 0; i < 10; i++) sum += v [i];
for ({sum = 0; i = 0;} i < 10; i++) sum += v [i];
```

7.7 Foreach-statement

This statement is used to execution of the foreach-statement body (the statement) for all keys of table which is a value of the expression. The expression value should be a table. If this is not true, the exception `keyop` is generated. The current key value on each iteration is assigned to the designator. The order in which the key values are assigned on each iteration is undefined. One iteration can be finished with the aid of the statement `continue` and a foreach-statement can be finished by execution of statement `break`.

```
ExecutiveStmt = for  "(" Designator in Expr ")" Stmt
```

Examples:

```
putln ("The table is");
for (k in t) {
  put ("key=");
  print (k);
  put (", element=");
  println (t{k});
}
```

7.8 Match-statements

There are two kinds of the match-statement. One is used for pattern matching and another one is used for regular expression matching. They have practically the same syntax. They differ in usage of different start keywords `pmatch` and `rmatch` correspondingly for the pattern and regular expression matching.

The pattern or regular expression match statement is used to try matching the match-expression value (it is evaluated only once) and the patterns or regular expressions in given order and execute the statements corresponding to the first matched case.

Each case forms an own scope. The pattern variables in the pattern match statement are declared in the corresponding case scope. Each case scope of the regular expression match statement contains an implicitly declared variable `m`. If a regular expression in the case successfully matches the match-expression, value of the corresponding variable `m` is a vector of indexes describing matched substrings (see the result of function `match` for details).

In the case of regular expression matching the string conversion is implicitly applied to the match-expression and the case expression values. The exception `optype` occurs if the conversion results are not strings. A case with wildcard `_` also can occur in a rmatch-statement. Matching with such case is always successful but the value of the variable `m` is undefined in this case.

Execution of a continue-statement in the case-statements results in continuing the process of matching the match-expression value with the subsequent patterns or regular expressions. Execution of a break-statement in the case-statements results in finishing the match-statement execution. There is an implicit break at the end of each case statement list.

If an optional case condition is given, then it is evaluated after the succesfull matching with the corresponding pattern and an arithmetic conversion is done to it. The value should be an integer, a long integer, or a floating-point number, otherwise the exception `optype` is generated. If the value is nonzero the all match is considered successfull, otherwise the subsequent case patterns are tried.

```
ExecutiveStmt = (pmatch | rmatch) "(" Expr ")" "{" CaseList "}"
CaseList = { case Pattern [CaseCond] ":" StmtList }
CaseCond = if Expr
```

Examples:

```
class c (a1, a2) {}
pmatch (c (2, 3)) {
  case c (i, j): putln (i, j);
  case _: putln ("default");
}
pmatch (c (2, 3)) {
  case c (i, j) if i == j: putln ("eq=", i, j);
  case c (i, j) if i != j: putln ("neq=", i, j);
  case _: putln ("default");
}

rmatch (str) {
  case "[a-zA-Z]+": putln ("word starting at ", m[0]);
  case "[0-9]+": putln ("number starting at ", m[0]);
  case _: putln ("anything else, m is undefined");
}
```

7.9 Break- and continue-statement

The statements `break` and `continue` are used correspondingly to finish execution of the closest-containing for-, foreach-, or match-statement covering the statement and to finish one iteration of the body of the for- or foreach-statement and to continue trying subsequent cases for the match-statement. These statement can be used only inside a for-, foreach-, or match-statement.

```
ExecutiveStmt = break ";"
              | continue ";"
```

Examples:

```
for (i = 0; i < 10; i++) {
   if (ind [i] < 0)
     continue;
   val = v [ind[i]];
}
for (i in t)
  if (t{i} == elval)
    break;
match (tree) {
  case leaf (n):
    putln ("leaf");
    if (n == 10) continue;
  case node (n1, n2):
    if (n1 != n2) break;
    putln ("special node");
  case _: putln ("might be a leaf with 10");
}
```

7.10 Return-statement

A return-statement is used to finish execution of a function, a fiber, or class block. The statement corresponds to the closest-containing function, fiber, or class covering the statement, so the return-statement can be placed only in a function, a fiber, or a class. The expression in a return-statement can be given only for functions. In this case, the expression value will be the value of the function call (instead of undefined value).

```
ExecutiveStmt = return  [ Expr ] ";"
```

Examples:

```
return;
return [10, 2:0]
```

7.11 Throw-statement

This statement generates an exception which is given by value of the expression. The expression should evaluate to an object of predeclared class `except` or of its sub-class. If this is not true, the exception `optype` is generated. How exceptions are processed is described in the following section.

```
ExecutiveStmt = throw  Expr ";"
```

Examples:

```
class myexcept (msg) {use error former msg;}
throw myexcept ("this is an user defined exception");
```

7.12 Try-block

Exceptions can be generated by the Dino interpreter when some conditions are not satisfied, by predeclared Dino functions, by other OS processes, by user interruptions, or by the user with the aid of a throw-statement. Actually, the exceptions are represented by an object of the predeclared class `except` or by an object of its sub-class. All predeclared exceptions are described in the section *Predeclared Identifiers*. To detect and process exceptions, a try-block can be used.

When an exception is generated, the closest-containing try-block which is covering the statement generating the exception or currently being executed (when this is is generated by an OS process or by an user interruption) is searched for. Then, expressions in the catch list elements are processed. The expression value in the catch list element being currently processed should be the predeclared class `except` or its sub-class. If the expression being processed is a class and the exception is an object of the class or an object of a sub-class of the class, the block corresponding to the given catch list element is executed. If there is no such catch expression, the closest-containing try-block covering the current try-block is searched for and processing the exception is repeated. If there are no more try-blocks, the program finishes with a diagnostic message which is dependent on the generated exception.

Blocks corresponding to catch list elements have a predeclared variable `e`. When the block execution starts, the variable contains the object representing the exception.

```
ExecutiveStmt = TryBlockStmt

TryBlockStmt = try Block { Catch }

Catch = catch  "(" ExceptClassList ")" Block

ExceptClassList = Expr { "," Expr }
```

Examples:

```
try {
  var ln;
  for (;;)
    ln = getln ();
} catch (eof) {
}
```

```
try {
  var v = [];
  v {1} = 0;
} catch (except) {
  put ("catching and propagating exception"); println (class (e));
  throw e;
}
```

7.13 Wait-statement

This statement is used for the synchronization of different threads in a Dino program. The expression can not contain a function, class, or a fiber call. The thread in which the statement has been executed waits until the expression value becomes nonzero. The expression value (after an implicit arithmetic conversion) should be an integer, a long integer, or a floating point number. Otherwise the exception `optype` is generated. When the expression value becomes nonzero, the statement after the expression (it is called a sync-statement) is executed without interruption by other threads. It is used as a critical region for the thread synchronization. In a critical region an execution of wait-statement is prohibited (it results in generation of the exception `syncwait`). Also fiber calls inside a critical region result in generation of the exception `syncthreadcall`.

```
ExecutiveStmt = wait  "(" Expr ")" Stmt
```

An example:

```
wait (!empty);
```

7.14 C code

All C code between pairs of brackets %{ and %} in one Dino file is *concatenated* in the same order as they occur in the file. The result code with some pre-appended C code providing an interface to the Dino interpreter internal data representation is compiled when the execution *the first time* achieves the location of the first %{ in the file. The result shared object is loaded and external variables and functions are searched lately in the same order as the shared objects are loaded.

```
ExecutiveStmt = C_CODE
```

If an error during the compilation or loading the shared object file occurs, the exception `compile` is generated.

An example:

```
%{
  #include <math.h>
  val_t isnan_p (int npars, val_t *vals) {
    val_t val;
    ER_node_t res = (ER_node_t) & val;

    ER_SET_MODE (res, ER_NM_int);
    ER_set_i (res, 0);
    if (npars == 1
        && ER_NODE_MODE ((ER_node_t) vals) == ER_NM_float
        && isnan (ER_f ((ER_node_t) vals)))
      ER_set_i (res, 1);
    return val;
  }
%}

extern isnan_p ();
putln (isnan_p (10.0));
```

8 Program

A Dino program is simply a sequence of statements. There is a special declaration useful for writing programs consisting of several files or for making Dino packages. This is an include-declaration. Before execution of any statements all include-declarations are replaced by files whose base names are given by the strings. It is made recursively, i.e. the files themselves can contain other include-declarations. There should be no infinite recursion in this. If + is present in the include-declaration, the file is inserted in any case. Without + the file is inserted only if it has been yet not inserted into the block of the declaration.

```
Program = StmtList

IncludeDeclaration = include ["+"] STRING ";"
```

Examples:

The following program outputs the first 24 Fibonachi numbers:

```
// Recursive function to compute Fibonacci numbers
fun fibonacci (n) {
     if (n <= 1) return 1;
     return (fibonacci(n-1) + fibonacci(n-2));
}

var i, fibnum;

fibnum = 0;
for (i = 0; i <= 24; i++) {
    fibnum = fibonacci(i);
    putln (i @ " " @ fibnum);
}
```

The following program outputs the number of prime numbers less than 8190:

```
var i, prime, k, count, flags;
var final SieveSize = 8190;

flags = [SieveSize + 1 : 0];
count = 0;
for (i = 0; i <= SieveSize; i++)
  flags[i] = 1;
```

```
for (i = 0; i <= SieveSize; i++)
  if (flags[i]) {
      prime = i + i + 3;
      k = i + prime;
      for (;1;) {
            if (k > SieveSize)
              break;
          flags[k] = 0;
          k += prime;
      }
      count++;
  }
println (count);
```

The following program outputs the number of occurrences of different numbers and identifiers in stdin:

```
var i, key, voc = tab[];
for (;;)
  try {
    var ln, a;

    ln = getln ();
    if (ln == "")
      continue;
    a = split (ln, "[^[:alnum:]]");
    for (i = 0; i < #a; i++)
      voc {a[i]} = (a[i] in voc ? voc [a[i]] + 1 : 1);
  } catch (eof) {
    break;
  }
fun comp (el1, el2) {
  return cmpv (tolower (el1), tolower (el2));
}
key = sort (keys (voc), comp);
for (i = 0; i < #key; i++)
  putln (key[i], " : ", voc[key[i]]);
```

The following program uses the Dino package `socket`:

```
include "socket";
var s, cl, str, l = 0;
s = sockets.stream_server (10003, 4);
cl = s.accept ();
try {
  for (;;) {
     str = cl.read (64); l += #str; cl.write (str);
  }
} catch (sockets.socket_eof_except) {
  putln ("i got ", l, " bytes");
}
```

9 Predeclared identifiers

Dino has quite a lot of predeclared identifiers. They are combined in in a few signleton objects also called spaces – see the section *Declarations and Scope Rules*. Most of predeclared identifiers refer for functions. The predeclared functions expect a given number of actual parameters (may be a variable number of parameters). If the actual parameter number is an unexpected one, the exception `parnumber` is generated. The predeclared functions expect that the actual parameters (may be after implicit conversions) are of the required type. If this is not true, the exception `partype` is generated. To show how many parameters the function requires, we will write the names of the parameters and use the brackets `[` and `]` for the optional parameters in the description of the functions.

Examples: The following description

```
            strtime ([format [, time]])
```

describes that the function can accept zero, one, or two parameters. If only one parameter is given, then this is parameter `format`.

If nothing is said about the returned result, the function return value is undefined.

The predeclared identifiers are describe below according to their spaces.

9.1 Space `lang`

The space contains fundamental Dino declarations. All declarations of the space are always exposed.

9.1.1 Predeclared variables

Space `lang` has some predeclared variables which contain useful information or can be used to control the behaviour of the Dino interpreter.

Arguments and environment To access arguments to the program and the environment, the following variables can be used:

- `argv`. The variable value is an immutable vector whose elements are strings (immutable vectors of characters) representing the arguments to the program (see the appendix *Implementation*).
- `env`. The variable value is an immutable table whose elements are strings (immutable vectors of characters) representing values of the environment variables whose names are the keys of the table.

Versions As Dino is a live programming language, it and its interpreter are in the process of development. To access the Dino interpreter's version number and the language version, the final variables `version` and `lang_version` can be used correspondingly. The variable values are the versions as floating point numbers. For example, if the current Dino interpreter version is 0.97 and the Dino language version is 0.5, the variable values will be 0.97 and 0.5.

Threads To access some information about threads in Dino program, the following variables can be used.

- `main_thread`. The variable value is the main thread. When the program starts, there is only one thread which is called *the main thread*.
- `curr_thread`. The variable value is the thread in which you reference for the variable.

All these variables are final, so you can not change their values.

9.1.2 Exception classes

All predeclared classes in the space `lang` describe exceptions which may be generated in a Dino program. All Dino exceptions are represented by objects of the predeclared class `except` or of a sub-class of the class `except`. The class `except` has no parameters. There is only one predeclared sub-class `error` of the class `except`. All classes corresponding to user-defined exceptions are suggested to be declared as a sub-class of `except`. All other exceptions (e.g. generated by the Dino interpreter itself or by predeclared functions) are objects of the class `error` or predeclared classes which are sub-classes of `error`. The class `error` and all its sub-classes has one parameter `msg` which contains a readable message about the exception. The following classes are declared in the space `lang` as a sub-class of `error`:

- `invop`. The following sub-classes of this class describe exceptions when operands of an operation have an incorrect type or value.
  - `optype`. This class describes that an operand of an operation is not of the required type (possibly after implicit conversions).
  - `opvalue`. This class is reserved for the error of that an operand of an operation has invalid value.
- `invindex`. Sub-classes of this class describe exceptions in referring for a vector element.
  - `indextype`. This class describes that the index is not of integer type (possibly after implicit integer conversion).
  - `indexvalue`. This class describes that the index is negative or equal to or more than the vector length.
  - `indexop`. This class describes that the first operand in referring to a vector element is not a vector.
- `invslice`. Sub-classes of this class describe exceptions in referring for a vector slice.
  - `slicetype`. This class describes that the start index, bound, or step is not of integer type (possibly after implicit integer conversion).
  - `sliceform`. This class describes that the slice has a wrong form, e.g. the start index is negative, the step is zero or the slice is applied not to a vector.
- `invector`. Sub-classes inside this class mostly describe exceptions in slice operations.
  - `veclen`. This class describes that operands in a slice operator have different length.
  - `vecform`. This class describes that operands in a slice operator have different dimensions.
  - `matrixform`. This class describes error when a matrix transposition (function `transpose`) is applied to a vector of different length vectors.

- `invkey`. Sub-classes inside this class describe exceptions in referring to a table element.
  - `keyvalue`. This class describes that there is no such element in the table with the given key when we need the value of the element. The exception does not occur when a table element reference stands in the left hand side of an assignment-statement.
  - `keyop`. This class describes that the first operand in referring to a table element is not a table.
- `invcall`. Sub-classes of this class describe exceptions in calling functions (mainly predeclared ones).
  - `abstrcall`. This class describes that we try to call a declared but not defined function.
  - `callop`. This class describes that we try to call something which is not a function, class, or fiber. The exception is also generated when we try to create a class `file` instance by calling the class.
  - `partype`. This class describes that a parameter value of a predeclared function is not of the required type.
  - `parvalue`. This class describes that a parameter value of a predeclared function is not one of the permitted values (see functions `set_encoding`, `set_file_encoding`).
  - `parnumber`. This class describes that the number of actual parameters is not valid when we call a predeclared function.
  - `syncthreadcall`. This class describes that a fiber call occurs inside a critical region – see the wait-statement.
  - `invresult`. This class describes that the result value of a function call is not of the required type, e.g. the comparison function used in a call of the function `sort` returns a non integer value.
  - `internal`. This class describes all other (nonspecified) exceptions in calling predeclared functions.
- `invaccess`. Sub-classes of this class describe exceptions in accessing or changing values.
  - `accessop`. This class describes that a given class declaration can not be found or is private when accessing to it through the corresponding object.
  - `accessvalue`. This class describes that we try to access to a declared but not defined through the corresponding object – see *abstract classes*.
  - `immutable`. This class describes that we try to change an immutable value.
  - `patternmatch`. This class describes that the pattern in a variable declaration does not match the assigned value.
- `deadlock`. This class describes that a deadlock is recognized in a multi-threaded program.
- `syncwait`. This class describes that we try to execute a wait-stmt inside a critical region.

9.1.3 Functions of the space `lang`

The following functions are declared in the space `lang`:

- `tolower (str)`. The function expects that the parameter `str` (after an implicit string conversion) is a string. The function returns a new string `str` in which upper case letters are changed to the corresponding lower case letters.

- `toupper (str)`. The function expects that the parameter `str` (after an implicit string conversion) is a string. The function returns a new string `str` in which lower case letters are changed to the corresponding upper case letters.
- `translit (str, what, subst)`. The function transliterates charactes in a string. The function expects that the parameters `str` (after an implicit string conversion), `what`, and `subst` are strings. The function returns the new string `str` in which its characters which are present in `what` are changed to the corresponding characters in `subst`. The last two strings should have the same length. The second string may contain more than one occurence of a character. In this case the last correspondence is taken.
- `eltype (vect)`. The function expects that the parameter value is a vector. The function returns `nil` if the vector is heterogenous, otherwise the function returns the type of the vector elements (type of `nil` if the vector is empty).
- `keys (tab)`. The function expects that the parameter value is a table. The function returns a new mutable vector containing all the keys in the table. The order of keys in the vector is undefined.
- `closure (par)`. The function accepts any parameter value. If the parameter value is an object or a block instance of a function, the function `closure` returns the corresponding class or function which contains also its context. That is why it is called a closure. In all other cases, the function returns `nil`.
- `context (par)`. The function returns the context (see the section *Declarations and Scope Rules*) represented by a block instance or an object for the given parameter value which should be a function, a class, a fiber, a block instance, or an object.
- `inside (par1, par2, flag = 0)`. The goal for the function usage is to check that something is declared inside something other. If the third parameter value after an implicit integer conversion is given and nonzero, it is checked with taking contexts into account. The second parameter value should be a function, class, object, or a block instance. In the last two cases of the second parameter value, the corresponding class, function, or block is used. The first parameter value should be a function, a class, an object, or a block instance. In the last two cases, they define the corresponding function, class, or block.

  If the function, class, or block defined by the first parameter is declared inside the function, class, or block given by the second parameter, the function `inside` returns 1. The function `inside` also returns 1 if the function, class, or block defined by the first parameter is the same as the function, class, or block given by the second parameter. Otherwise the function `inside` returns 0. The following example illustrates the difference between checking with taking contexts into account and without it.

  ```
  class c () {
    class subc () {
    }
  }
  inside (c ().subc (), c ().subc);  // returns 1
  inside (c ().subc (), c ().subc, 1); // returns 0
  ```

  The first call of `inside` returns 1, while the second one returns 0.
- `isa (fco, fc)`. The goal for function usage is to check that a function, a class, or an object given by the first parameter `fco` uses declarations (through a use-clause) of a function or a class given by the second parameter `fc`, in other words the first is a subtype of the second (or a sub-class of the class).

If it is true, the function returns 1, otherwise it returns zero. If the parameter types are wrong, the function generates the exception `partype`. The following example illustrates usage of `isa`.

```
class c () {}
class subc () { use c;}
isa (subc, c);
isa (subc (), c);
```

The calls of `isa` in the example return 1.

- `subv (vect, index, length = -1)`. The function is used to extract a sub-vector. The first parameter value should be a vector after an implicit string conversion. The second and third parameter values should be integers after an implicit integer conversion. The function extracts only an element or the part of the sub-vector existing in the vector (so you can use any values of the index and the length). If the index is negative, it refers to an element anologous to a slice bound. In other words, `-1` corresponds to the vector length, `-2` corresponds to the vector length-1, `-3` corresponds to the vector length-2, and so on. If the length is negative, the sub-vector will finish on the vector end. The function returns a new vector which is the sub-vector. The result vector is immutable only when the original vector is immutable.

- `del (vect, index, length = 1) or del (tab, key)`. The first form of the function is used to remove the vector element or a sub-vector from the mutable vector `vect`. The second and the third parameter values should be integers after an implicit integer conversion. The function removes only an element or the part of the sub-vector existing in the vector (so you can use any values of the index and the length). A negative index has the same meaning as in `subv`. If the length is negative, the sub-vector will finish on the vector end.

  The second form of the function is used to remove an element (if it exists) with the given key from a mutable table.

  The function generates the exception `immutable` if we are trying to remove from an immutable vector or table. The function returns the modified vector/table.

- `ins (vect, el, index = -1)`. The function inserts an element given by the second parameter into a vector given by the first parameter on the place given by the third parameter. The third parameter should be an integer after an implicit integer conversion. Negative index has the same meaning as in `subv`. The function generates the exception `immutable` if we are trying to insert into an immutable vector. The function returns the modified vector.

- `insv (vect, vect, index = -1)`. The function is analogous to the function `ins` but it is used for insertion of all vector elements into the vector given as the first parameter. So the second parameter value should be a vector. The function returns the modified vector.

- `rev (vect)`. The function returns a reversion of the given vector.

- `cmpv (vect, vect)`. The function makes ab implicit string conversion of the parameter values. After that, the parameter values should be vectors whose first corresponding equal elements should have the same type (character, integer, or floating point type). The first corresponding unequal elements should have the same type too (the remaining elements can have different types). As usual, if this is not true, the exception `partype` is generated. The function returns 1 if the first unequal element value of the first vector is greater than the corresponding element in the second vector, -1 if less, and 0 if the all corresponding vector elements are equal. If the first vector is a prefix of the second vector, the function returns -1. If the second vector is a prefix of the first vector, the function returns 1, so it uses in fact a generalized lexicographical order.

- `filter (f, v, d = 1)`. The function expects function `f`, vector `v`, and optional integer `d` after an integer conversion. Otherwise the exception `partype` is generated. The function processes v's elements if `d` is equal one, elements of vectors which are v's elements if `d` is equal to 2 and so on. In other words, `d` is a level on which the vector elements are processed. If `v` has no structure necessary for processing, the exception `vecform` is generated. If `d` is zero or negative, the function just returns `v`. Otherwise the function creates a new mutable vector having the same structure as `v` with only elements on level `d` for which the function `f` returns nonzero value after an integer conversion.

  If the result of function `f` calls after the integer conversion is not integer, the exception `invresult` is generated. The following example illustrates an usage of `filter`.

```
var i, v = [0, 1, -2, 3, -4];
println (filter (fun (a) {a > 0;}, v));
v = [[0, 1, -2, 3, -4], [5, -6, 7, -8, 9]];
println (filter (fun (a) {a > 0;}, v, 2));
```

- `map (f, v, d = 1)`. The meaning of the function parameters and constraints to their values are analogous to ones of the function `filter`. Only the function `f` can return any value. The elements processed by the function `f` are changed onto the results of function `f` calls. The following example illustrates usage of `map`.

```
var i, v = [[0, 1, -2, 3, -4], [5, -6, 7, -8, 9]];
println (map (fun (a) {a < 0 ? nil : a;}, v, 2));
```

- `fold (f, v, init, d = 1)`. The meaning of function parameters `f`, `v`, and `d` and constraints to their values are analogous to ones of the function `filter`. The function processes all elements of the vectors on level `d` and returns value `f (f (f (init, el0), el1), ...)  , eln)` where `el0`, ..., `eln` are vector elements on level `d` taken from left to right. If `d` is zero or negative or the vectors are empty, the function returns `init`. The following example illustrates usage of `fold`.

```
var v = [1,2,3,4];
println (fold (fun (a, b) {a + b;}, v, 0));
```

- `sort (vect[, compare_function])`. The function returns a new sorted vector. The original vector given as the first parameter value should be a homogeneous vector whose elements are of character, integer, long integer, or floating point type. If the second parameter is not given, the standard arithmetic order (see the comparison operators) is used. To use a special ordering, use the second parameter which should be a function which compares two elements of the vector and returns a negative integer if the first parameter value (element) is less than the second one, a positive integer if the first parameter value is greater than the second one, and zero if they are equal.

- `transpose (m)`. The function expects matrix `m`. It means that `m` should be a vector (each element is a matrix row) of vectors of equal length. If `m` is not a vector, the exception `partype` is generated. If the elements of `m` are not vectors of the same length, the exceptions `matrixform` is generated. The function returns a new mutable vector of mutable vectors which is a matrix transposition of `m`.

- `gc ()`. The function forces a garbage collection and heap compaction. Usually the Dino interpreter itself invokes a garbage collection when it believes that it needs to this.

- `exit (code)`. The function finishes the work of the interpreter with the given code which should be an integer value after an implicit integer conversion.

9.2 Space `io`

The space contains functions for input and output and for work with files and directories. All declarations of the space are always exposed.

9.2.1 Exception classes of the space `io`

The following classes are declared in the space `io` as sub-classes of `invcall`:

- `invinput`. This class describes that the file input is not of the required format. Usually the exception is generated by the function `scan` etc.
- `invfmt`. This class describes that a format of a format output function is wrong (see the function `putf`).
- `eof`. This class describes that the end of file is encountered. Usually the exception is generated by functions reading files (`get`, `scan` etc).
- `invencoding`. This class describes different exceptions with the used encodings, e.g. a file contains bytes not corresponding to the expected encoding or in some cases the encoding should contain ASCII characters.

9.2.2 Class `file`

Dino has a predeclared final class `file`. Work with files in a Dino program are made through objects of the class. All declarations inside of the class are private. The objects of the class can be created only by the predeclared functions `open` or `popen`. If you create an object of the class by calling the class, the exception `callop` will be generated. The file encoding is defined by the current DINO encoding at the file creation time (see thefunctions `set_encoding`, `set_file_encoding`). If you want to work with files on the byte level without any encoding/decoding, you can use an encoding called `"RAW"`.

9.2.3 Files

To output something into the standard output streams or to input something from the standard input stream, the following variables can be used:

- `stdin`. The variable value is an object of the class `file` which corresponds to the standard input stream.
- `stdout`. The variable value is an object of the class `file` which corresponds to the standard output stream.
- `stderr`. The variable value is an object of the class `file` which corresponds to the standard error stream.

All these variables are final, so you can not change their values. Encoding of the files is DINO current encoding at the program start (see the function `set_encoding`).

9.2.4 Functions for work with files

The following functions (besides the input/output functions) work with OS files. The functions may generate an exception declared in the class `syserror` (e.g. `eaccess`, `enametoolong`, `eisdir` and so on) besides the standard `partype`, and `parnumber`. The function `rename` can be used for renaming a directory, not only a file.

- `rename (old_path, new_path)`. The function renames the file (directory) given by its path name. The old and new names are given by the parameter values which should be strings after an implicit string conversion.
- `remove (file_path)`. The function removes the OS file given by its path name. The file path name should be a string after an implicit string conversion.
- `open (file_path, mode)`. The function opens the file for work in the given *mode*, creates a new class `file` instance, associates the opened file with the instance, and returns the instance. The parameter values should be strings after an implicit string conversions. The first parameter value is a string representing the file path. The second parameter value is a string representing the mode for work with the file (for all possible modes see the ANSI C function `fopen` documentation). All work with the opened file is made through the file instance.
- `close (fileinstance)`. The function closes a file opened by the function `open`. The file is given by the class `file` instance. The function also removes all association of the instance with the file.
- `flush (fileinstance)`. The function flushes any output that has been buffered for the opened file given by the class `file` instance.
- `popen (command, mode)`. The function starts a shell command given by the first parameter value (which should be a string after an implicit string conversion), creates a pipe, creates a new class `file` instance, associates the pipe with the instance, and returns the instance. Writing to such a pipe (through the class file instance) writes to the standard input of the command. Conversely, reading from the pipe reads the command's standard output. After an implicit string conversion the second parameter value should be the string `"r"` (for reading from the pipe) or `"w"` (for writing to the pipe). The pipe should be closed by the function `pclose`.
- `pclose (fileinstance)`. The function waits for the command connected to a pipe to terminate. The pipe is given by the class `file` instance returned by the function `popen`. The function also removes the association of the instance with the pipe.
- `tell (fileinstance)`. The function returns the current value of the file position indicator for the file (opened by function `open`) given by the class `file` instance.
- `seek (fileinstance, offset, whence)`. The function sets up the current file position indicator for the file (opened by function `open`) given by the class `file` instance. The position is given by `offset` which should be an integer after an implicit arithmetic conversion and `whence` which should be a string after an implicit string conversion. The first character of the string should be `'s'`, `'c'`, or `'e'` (these characters mean that the offset is relative to the start of the file, the current position indicator, or the end-of-file, respectively).
- `get_file_encoding (fileinstance)`. The function returns a new mutable string which is a name of the current file encoding.

- `set_file_encoding (fileinstance, name)`. The function accepts a file and a string and changes the current file encoding. If the name represents an unknown encoding name, the function generates the exception `parvalue`.

9.2.5 File output functions

The following functions are used to output something into opened files. All the function return values are undefined. The functions may generate an exception declared in the class `syserror` (e.g. `eio`, `enospc` and so on) besides the standard `partype` and `parnumber`.

- `put (...)`. All parameters should be strings after an implicit string conversion. The function outputs all strings into the standard output stream.
- `putln (...)`. The function is analogous to the function `put` except for the fact that it additionally outputs a new line character after output of all the strings.
- `fput (fileinstance, ...)`. The function is analogous to the function `put` except for the fact that it outputs the string into an opened file associated with a class `file` instance which is the first parameter value.
- `fputln (fileinstance, ...)`. The function is analogous to function `fput` except for the fact that it additionally outputs a new line character after output of all the strings.
- `putf (format, ...)`. The first parameter should be a string after an implicit string conversion. The function outputs the rest of parameters according to the format. The number of the rest parameters should be exactly equal to the conversions (including parameterized widths and precisions) in the format. Otherwise, the exception `parnumber` will be generated. The types of the parameter should correspond to the corresponding conversion specifier (or to be an integer for parameterized widths and precisions). If it is not true, the exception `partype` will be generated. The format is mostly a subset of one of standard C function `printf` but it can also deal with multi-precision integers (of the Dino type long). The format has the following syntax:

```
format : <any character except %>
       | '%' flags [width] [precision]
             conversion_specifier
flags :
      | flag

flag : '#' | '0' | '-' | ' ' | '+'

width : '*' | <a decimal number starting with non-zero>

precision : '.' ['*' | <decimal number>]

conversion_specifier : 'd' | 'o' | 'x' | 'X'
                     | 'e' | 'E' | 'f' | 'g'
                     | 'G' | 'c' | 's' | '%'
```

  If the format syntax is wrong, the exception `invfmt` is generated.

The flag '#' means that the value should be converted into an alternative form. It can be present only for the conversion specifiers 'o', 'x', 'X', 'e', 'E', 'f', 'g', and 'G'. If the flag is used for the conversion specifier 'o', the output will be prefixed by '0'. For 'x' and 'X' the output will be prefixed by '0x' and '0X' correspondingly. For the conversions 'e', 'E', 'f', 'g', and 'G' the output will always contain a decimal point. For the conversions 'g' and 'G' it also means that trailing zeros are not removed from the output as they would be without the flag. The following code using the flag '#' in a format

```
putf ("->%#o %#x %#x %#.0e %#.0f %#g<-\n",
      8, 10, 16l, 2., 3., 4.);
```

will output

```
->010 0xa 0x10 2.e+00 3. 4.00000<-
```

The flag '0' means that the output value will be zero padded on the left. If both flags '0' and '-' appear, the flag '0' is ignored. It is also ignored for the conversions 'd', 'o', 'x', and 'X' if a precision is given. The flag is prohibited for the conversions 'c' and 's'. The following code using the flag '0' in a format

```
putf ("->%04d %04x %04x %09.2e %05.2f %05.2g<-\n",
      8, 10, 16l, 2., 3., 4.);
```

will output

```
->0008 000a 0010 02.00e+00 03.00 00004<-
```

The flag '-' means that the output will be left adjusted on the field boundary. (The default is a justification to the right). The flag '-' overrides the flag '0' if the both are given. The following code using the flag '-' in a format

```
putf ("->%-04d %-04x %-04x %-09.2e %-05.2f %-05.2g<-\n",
      8, 10, 16l, 2., 3., 4.);
```

will output

```
->8    a    10   2.00e+00  3.00  4    <-
```

The flag ' ' means that the output of a signed number will start with a blank for positives number. The flag can be used only for the conversions 'd', 'e', 'E', 'f', 'g', and 'G'. If both flags ' ' and '+' appear, the flag ' ' is ignored. The following code using the flag ' ' in a format

```
putf ("->% d % d % .2e % .2f % .2g<-\n",
      8, 16l, 2., 3., 4.);
```

will output

```
-> 8  16  2.00e+00  3.00  4<-
```

The flag '+' means that the output of a signed number will start with a plus for a positives number. The flag can be used only for the conversions 'd', 'e', 'E', 'f', 'g', and 'G'. The flag '+' overrides the flag ' ' if both are given. The following code using the flag '+' in a format

```
putf ("->%+d %+d %+.2e %+.2f %+.2g<-\n",
      8, 16l, 2., 3., 4.);
```

will output

```
->+8 +16 +2.00e+00 +3.00 +4<-
```

The width defines a minimum width of the output value. If the output is smaller, it is padded with spaces (or zeros – see the flag '0') on the left (if the flag '-' is used) or on the right. The output is never truncated. The width should be no more than maximal integer value, otherwise teh exception `invfmt` is generated. The width can be given as a parameter of the integer type if '*' is used. If the value of the width given by the parameter is negative, the flag '-' is believed to be given and the width is believed to be equal to zero. The following code using the width in a format

```
putf ("->%5d %05d %-5d %5d %*d %*d<-\n",
      8, 9, 10, 16l, 5, 8, -5, 10);
```

will output

```
->    8 00009 10       16     8 10   <-
```

The precision is prohibited for the conversion 'c'. If the number after the period is absent, its value will be zero. The precision can be given as a parameter of the integer type if '*' is used after the period. If the value of precision given by the parameter is negative, its value is believed to be zero too. For the conversions 'd', 'o', 'x', and 'X' the precision means a minimum number of the output digits. For the conversions 'e', 'E', and 'f' it means the number of the digits to appear after the decimal point. For 'g' and 'G' it means the maximum number of significant digits. For 's' it means the maximum number of characters to be output from a string. The following code using precisions in a format

```
putf ("->%.d %.0d %.5d %.d %.0f %.0e %.2g<-\n",
      8, 8, 9, 16l, 2.3, 2.3, 3.53);
putf ("->%.2s %.0d %.*d %.*d %.*d<-\n",
      "long", 0, 5, 8, -5, 8, 5, 16l);
```

will output

```
->8 8 00009 16 2 2e+00 3.5<-
->lo  00008 8 00016<-
```

The conversion 'd' should be used to output integer or long integer. The default precision is 1. When 0 is output with an explicit precision equal to 0, the output is empty.

The conversions 'o', 'x', and 'X' should be used to output an integer or long integer value as an unsigned in the octal and hexadecimal form. The lower case letters `abcdef` are used for 'x' and the upper case letters `ABCDEF` are used for 'X'. The precision gives the minimum number of digits that must appear. If the output value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is output with an explicit precision equal to 0, the output is empty.

The conversion 'f' should be used to output floating point values. The output value has a form `[-]ddd.ddd` where the number of digits after the decimal point is given by the precision specification. The default precision value is 6. If the precision is explicitly zero, no decimal-point character appears.

The conversions 'e' and 'E' should be used to output floating point values with an exponent in the form `[-]d.ddd[e|E][+|-]dd`. There is always one digit before the decimal-point. The number of digits after the decimal point is defined by the precision. The default precision value is 6. If the precision is zero, no decimal-point appears. The conversion 'E' uses the letter `E` (rather than `e`) to introduce the exponent. The exponent always contains at least two digits. If the exponent value is zero, the exponent is output as `00`.

The conversions 'g' and 'G' should be used to output floating point values in the style 'f' or 'e' (or 'E' for conversion 'G'). The precision defines the number of significant digits. The default value of the precision is 6. If the precision is zero, it is treated as 1. The conversion 'e' is used if the exponent from the conversion is less than -4 or not less than the precision. Trailing zeros are removed from the fractional part of the output. If all fractional part is zero, the decimal point is removed too.

The conversion 'c' should be used to output a character value.

The conversion 's' should be used to output strings.

The conversion '%' should be used to output `%`.

The following code using different conversions in a format

```
putf ("->%% %c %s %d %o %x %X %d %o %x %X<-\n",
      'c', "string", 7, 8, 20, 20, 8l, 9l, 21l, 21l);
putf ("->%f<-\n", 1.5);
putf ("->%e %E %g %G %g %G<-\n",
      2.8, 2.8, 3.7, 3.7, 455555555.555, 5.9e-5);
```

will output

```
->% c string 7 10 14 14 8 11 15 15<-
->1.500000<-
->2.800000e+00 2.800000E+00 3.7 3.7 4.55556e+08 5.9E-05<-
```

- `fput (fileinstance, format, ...)`. The function is analogous to the function `putf` except for the fact that it outputs the operands into an opened file associated with a class `file` instance which is the first parameter value.
- `print (...)`. The function outputs all parameter values into the standard output stream. The function never makes an implicit conversions of the parameter values. The parameter values are output as they could be represented in Dino itself (e.g. character `'c'` is output as `'c'`, vector `['a', 'b', 'c']` is output as `"abc"`, vector `[10, 20]` as `[10, 20]` and so on). As you know some values (functions, classes, block instances, class instances, threads) are not represented fully in DINO. Such values are represented schematically. For example, the output `fun f {}.g(unique_number)` would mean the function `f` in the call of function (or class) `g` with the given unique number and the function g is in the instance of the implicit block covering the whole program. For the function `g`, output would look simply like `fun g` because there is only one instance of the implicit block covering the whole program. Output for an instance of the class `c` in the function `f` looks like `instance {}.f(unique_number).c(unique_number)`. Output for a block instance of the function `f` looks like `stack {}.f(unique_number)`. Output for a thread whose fiber `t` is declared in the function `f` would look like `thread unique_number {}.f(unique_number).t(unique_number)`.
- `println (...)`. The function is analogous to the function `print` except for the fact that it additionally outputs a new line character after output of all parameters.
- `fprint (fileinstance, ...)`. The function is analogous to the function `print` except for the fact that it outputs the parameters into an opened file associated with a class `file` instance which is the value of first parameter.
- `fprintln (fileinstance, ...)`. The function is analogous to function `fprint` except for the fact that it additionally outputs a new line character after the output of all the parameters.

9.2.6 File input functions

The following functions are used to input something from opened files. The functions may generate an exception declared in the class `syserror` (e.g. `eio`, `enospc` and so on) or `eof` besides the standard `partype`, and `parnumber`.

- `get ()`. The function reads one character from the standard input stream and returns it. The function generates the exception `eof` if the function tries to read the end of file.
- `getln ()`. The function reads one line from the standard input stream and returns it as a new string. The end of line is the newline character or end of file. The returned string does not contain the newline character. The function generates the exception `eof` only when the file position indicator before the function call stands exactly at the end of file.
- `getf ([ln_flag])`. The function reads the whole standard input stream and returns it as a new string. The function generates the exception `eof` only when the file position indicator before the function call stands exactly at the end of file. The function has an optional parameter which should be integer after an implicit integer conversion. If the parameter value is nonzero, the function returns a vector of strings. Otherwise it behaves as usually. Each string is a line in the input stream. The strings do not contain the newline characters.
- `fget (fileinstance)`. The function is analogous to the function `get` except for the fact that it reads from an opened file associated with the class `file` instance which is the parameter's value.
- `fgetln (fileinstance)`. The function is analogous to the function `getln` except for the fact that it reads from an opened file associated with a class `file` instance which is the parameter value.
- `fgetf (fileinstance [, ln_flag])`. The function is analogous to the function `getf` except for the fact that it reads from an opened file associated with a class `file` instance which is the parameter's value.
- `scan ()`. The functions reads a character, integer, floating point number, string, vector, or table and returns it as the result. The input values should be represented in the file as the ones in the Dino language (except for the fact that there should be no identifiers in the input values and there should be no operators in the values, although the signs `+` and `-` are possible in an integer or floating point represenation). The table or vector should contains only values of the types mentioned above. The values in the file can be separated by white characters. If there is an error (e.g. unbalanced brackets in a vector value) in the read value representation the function generates the exception `invinput`. The functions generates the exception `eof` if only white characters are still unread in the file.
- `scanln ()`. The function is analogous to the function `scan` except for the fact that it skips all characters until the end of line or the end of file after reading the value. Skipping is made even if the exception `invinput` is generated.
- `fscan (fileinstance)`. The function is analogous to the function `scan` except for the fact that it reads from an opened file associated with a class `file` instance which is the parameter's value.
- `fscanln (fileinstance)`. The function is analogous to the function `scanln` except for that it reads from an opened file associated with a class `file` instance which is the parameter value.

9.2.7 Encoding functions

Dino internally uses Unicode for characters. To provide a communication with the rest of world, it can use different encodings. The default encoding is UTF-8. Dino has two functions to get and change the current encoding:

- `get_encoding ()`. The function returns a new mutable string which is a name of the current encoding.
- `set_encoding (name)`. The function accepts a string and changes the current encoding. If the name represents an unknown encoding name, the function generates the exception `parvalue`.

Examples:

```
putln (get_encoding ());
set_encoding ("KOI8-R");
```

9.2.8 Functions for work with directories

The following functions work with directories. The functions may generate an exception declared in the class `syserror` (e.g. `eaccess`, `enametoolong`, `enotdir` and so on) besides the standard `partype` and `parnumber`.

- `readdir (dirpath)`. The function makes an implicit string conversion of the parameter value which should be a string (representing a directory path). The function returns a new mutable vector with elements which are strings representing the names of all files and sub-directories (including `"."` and `".."` for the current and parent directory respectively) in given directory.
- `mkdir (dirpath)`. The function creates a directory with the given name represented by a string (the parameter value after an implicit string conversion). The directory has read/write/execute rights for all. You can change it with the aid of the functions `ch*mod`.
- `rmdir (dirpath)`. The function removes the directory given by a string which is a parameter value after an implicit string conversion.
- `getcwd ()`. The function returns a new string representing the full path of the current directory.
- `chdir (dirpath)`. The function makes the directory given by `dirpath` (which should be a string after an implicit string conversion) the current directory.

9.2.9 Functions for access to file/directory information

The following predeclared functions can be used for accessing file or directory information. The functions may generate an exception declared in the class `syserror` (e.g. `eaccess`, `enametoolong`, `enfile` and so on) besides the standard `partype` and `parnumber`. The functions expect one parameter which should be a file instance (see the predeclared class `file`) or the path name of a file represented by a string (the functions make an implicit string conversion of the parameter value). The single exception to this is `isatty` which expects a file instance.

- `ftype (fileinstance_or_filename)`. The function returns one the following characters:
  - `'f'`. A regular file.

  - `'d'`. A directory.
  - `'L'`. A symbolic link.
  - `'c'`. A character device.
  - `'b'`. A block device.
  - `'p'`. A fifo.
  - `'S'`. A socket.

  Under some OSes the function never returns some of the characters (e.g. 'c' or 'b'). The function may return nil if it can not categorize the file as above.

- `fuidn (fileinstance_or_filename)`. The function returns a new string representing a name of the owner of the file (directory). Under some OSes the function may return the new string `"Unknown"` if there is no notion "owner" in the OS file system.

- `fgrpn (fileinstance_or_filename)`. Analogous to the previous function except for it returns a new string representing a name of the group of the file (directory). Under some OSes the function may return the new string `"Unknown"` if there is no notion "group" in the OS file system.

- `fsize (fileinstance_or_filename)`. The function returns an integer value which is the length of the file in bytes.

- `fatime (fileinstance_or_filename)`. The function returns an integer value which is time of the last access to the file (directory). The time is measured in seconds since the fixed time (usually since January 1, 1970). See also *time functions*.

- `fmtime (fileinstance_or_filename)`. Analogous to the previous functions but returns the time of the last modification.

- `fctime (fileinstance_or_filename)`. Analogous to the previous functions but it returns the time of the last change. Here 'change' usually means changing the file attributes (owner, modes and so on), while 'modification' means usually changing the file itself.

- `fumode (fileinstance_or_filename)`. The function returns a new string representing the rights of the owner of the file (directory). The string may contain the following characters (in the following order if the string contains more than one character):

  - `'s'`. Sticky bit of the file (directory).
  - `'r'`. Right to read.
  - `'w'`. Right to write.
  - `'x'`. Right to execute.

- `fgmode (fileinstance_or_filename)`. Analogous to the previous function except for the fact that it returns information about the file (directory) group user rights and that the function never returns a string containing the character `'s'`.

- `fomode (fileinstance_or_filename)`. Analogous to the previous function except for the act that it returns information about the rights of all other users.

- `isatty (fileinstance)`. The function returns 1 if the file instance given as the parameter is an open file connected to a terminal and 0 otherwise.

The following functions can be used to change the rights of usage of the file (directory) for different users. The function expects two strings (after an implicit string conversion). The first one is the path name of the file (directory). The second one is the rights. For instance, if the string contains the character 'r', this is a right to read (see characters used to denote different rights in the description of the function `fumode`). The function return values are always undefined.

- `chumod (path, mode)`. The function sets up rights for the file (directory) owner according to the given mode.
- `chgmod (path, mode)`. Analogous to the previous function except for the fact that it sets up rights for the file (directory) group users and that the function ignores the character `'s'`.
- `chomod (path, mode)`. Analogous to the previous function except for the fact that it sets up rights for all other users.

9.2.10 Miscellaneous functions

There are the following miscellaneous functions in space `io`:

- `sput (...), sputln (...), sputf (format, ...)` The functions are analogous to the functions `put, putln, print`, and `println` but they return the result string instead of output of the formed string into the standard output stream.
- `sprint (...), sprintln (...)`. The functions are analogous to the functions `print` and `println` but they return the result string instead of output of the formed string into the standard output stream.

9.3 Space sys

This space contains declarations to work with the underlying execution environment (OS) and related exceptions.

9.3.1 Exceptions in space sys

The space contains a lot of exceptions:

- `signal`. This class is a sub-class of the class `error`. Sub-classes of the class `signal` describe exceptions from receiving a signal from other OS processes. They are
  - `sigint`. This class describes the exception generated by the user's interrupt from the keyboard.
  - `sigill`. This class describes the exception generated by illegal execution of an instruction .
  - `sigabrt`. This class describes the exception generated by the signal abort.
  - `sigfpe`. This class describes a floating point exception.
  - `sigterm`. This class describes the exception generated by the termination signal.
  - `sigsegv`. This class describes the exception generated by an invalid memory reference.
- `invenv`. This class is a sub-class of the class `error`. The class `invenv` describes a corruption of the Dino program environment (see the predeclared variable `env`).

- `syserror`. This class is a sub-class of the class `invcall`. Sub-classes of the class `syserror` describe exceptions in predeclared functions which call OS system functions. Some exceptions are never generated but may be generated in the future on some OSes.
  - `eaccess`. This describes the system error "Permission denied".
  - `eagain`. This describes the system error "Resource temporarily unavailable".
  - `ebadf`. This describes the system error "Bad file descriptor".
  - `ebusy`. This describes the system error "Resource busy".
  - `echild`. This describes the system error "No child processes".
  - `edeadlk`. This describes the system error "Resource deadlock avoided".
  - `edom`. This describes the system error "Domain error".
  - `eexist`. This describes the system error "File exists".
  - `efault`. This describes the system error "Bad address".
  - `efbig`. This describes the system error "File too large".
  - `eintr`. This describes the system error "Interrupted function call".
  - `einval`. This describes the system error "Invalid argument".
  - `eio`. This describes the system error "Input/output error".
  - `eisdir`. This describes the system error "Is a directory".
  - `emfile`. This describes the system error "Too many open files".
  - `emlink`. This describes the system error "Too many links".
  - `enametoolong`. This describes the system error "Filename too long".
  - `enfile`. This describes the system error "Too many open files in system".
  - `enodev`. This describes the system error "No such device".
  - `enoent`. This describes the system error "No such file or directory".
  - `enoexec`. This describes the system error "Exec format error".
  - `enolck`. This describes the system error "No locks available".
  - `enomem`. This describes the system error "Not enough space".
  - `enospc`. This describes the system error "No space left on device".
  - `enosys`. This describes the system error "Function not implemented".
  - `enotdir`. This describes the system error "Not a directory".
  - `enotempty`. This describes the system error "Directory not empty".
  - `enotty`. This describes the system error "Inappropriate I/O control operation".
  - `enxio`. This describes the system error "No such device or address".
  - `eperm`. This describes the system error "Operation not permitted".
  - `epipe`. This describes the system error "Broken pipe".
  - `erange`. This describes the system error "Result too large".
  - `erofs`. This describes the system error "Read-only file system".
  - `espipe`. This describes the system error "Invalid seek".

  - `esrch`. This describes the system error "No such process".
  - `exdev`. This describes the system error "Improper link".

- `systemcall`. This is a sub-class of the class `invcall`. Sub-classes of the class `systemcall` describe exceptions in calling the predeclared function `system`.
  - `noshell`. This class describes the exception that the function `system` can not find the OS command interpreter (the shell).
  - `systemfail`. This class describes all remaining exceptions in calling the OS function `system`.
- `invextern`. This is a sub-class of the class `invcall`. Sub-classes of the class `invextern` describe exceptions in calling external functions or in accessing an external variable.
  - `noextern`. This class describes the exception that the given external can not be found.
  - `libclose`. This class describes the exception that there is an error in closing a shared library.
  - `noexternsupp`. This class describes an exception in the usage of external objects when they are not implemented under this OS.
  - `compile`. This class describes an exception in a compilation of C code or loading the result shared object file.
- `invenvar`. This is a sub-class of the class `invcall`. The class `invenvar` describes corruption in the type of variables `split_regex` and `time_format` (e.g. their values are not strings).

9.3.2 Variable `time_format`

The variable value is a string which is the output format of time used by the function `strtime` when it is called without parameters. The initial value of the variable is the string `"%a %b %d %H:%M:%S %Z %Y"`.

9.3.3 Time functions

The following functions from the space `sys` can be used to get information about real time.

- `time ()`. The function returns the time in seconds since the fixed time (usually since January 1, 1970).
- `strtime ([format [, time]])`. The function returns a string representing the `time` (an integer representing time in seconds since the fixed time) according to the `format` (a string). If the format is not given, the value of the variable `time_format` is used. In this case if the value of `time_format` is corrupted (it is not a string), the function generates the exception `invenvar`. If the time is not given, the current time is used. The format is the same as in C library function `strftime`. Here is an extraction from the OS function documentation. The following format specifiers can be used in the format:
  - `%a` - the abbreviated weekday name according to the current locale.
  - `%A` - the full weekday name according to the current locale.
  - `%b` - the abbreviated month name according to the current locale.
  - `%B` - the full month name according to the current locale.
  - `%c` - the preferred date and time representation for the current locale.

- `%d` - the day of the month as a decimal number (range 01 to 31).
- `%H` - the hour as a decimal number using a 24-hour clock (range 00 to 23).
- `%I` - the hour as a decimal number using a 12-hour clock (range 01 to 12).
- `%j` - the day of the year as a decimal number (range 001 to 366).
- `%m` - the month as a decimal number (range 01 to 12).
- `%M` - the minute as a decimal number.
- `%p` - either 'am' or 'pm' according to the given time value, or the corresponding strings for the current locale.
- `%S` - the second as a decimal number.
- `%U` - the week number of the current year as a decimal number, starting with the first Sunday as the first day of the first week.
- `%W` - the week number of the current year as a decimal number, starting with the first Monday as the first day of the first week.
- `%w` - the day of the week as a decimal, Sunday being 0.
- `%x` - the preferred date representation for the current locale without the time.
- `%X` - the preferred time representation for the current locale without the date.
- `%y` - the year as a decimal number without a century (range 00 to 99).
- `%Y` - the year as a decimal number including the century.
- `%Z` - the time zone or the name or an abbreviation.
- `%%` - the character '%'.

9.3.4 Functions for access to information about OS processes

The space `sys` contains predeclared functions which are used to get information about the current OS process (the Dino interpreter which executes the program). Each OS process has unique identifier and usually the OS processes are called by a concrete user and group and are executed on behalf of the concrete user and group (so called effective identifiers). The following functions return such information. On some OSes the function may return string "Unknown" as a name if there are no notions of user and group identifiers.

- `getpid ()`. The function returns an integer value which is the process ID of the current OS process.
- `getun ()`. The function returns a new string which is the user name for the current OS process.
- `geteun ()`. The function returns a new string which is the effective user name for the current OS process.
- `getgn ()`. The function returns a new string which is the group name for the current OS process.
- `getegn ()`. The function returns a new string which is the effective group name for the current OS process.
- `getgroups ()`. The function returns a new vector of strings (possibly the empty vector) representing supplementary group names for the current OS process.

9.3.5 Function `system (command)`

The function executes the command given by a string (the parameter value) in the OS command interpreter. Besides the standard exceptions `parnumber` and `partype` the function may generate the exceptions `noshell` and `systemfail`.

9.4 Space `re`

This space contains declarations which can be useful for working with the regular expressions and for pattern matching – see also the *match-statements*.

9.4.1 Exception class `invregex`

This class describes exceptions specific for executing the *pmatch-statement* and for calling predeclared functions implementing regular expression pattern matching. Although there is only one class for this, the messages which are in the class parameter can be different and explain more details.

9.4.2 Variable `split_regex`

The variable value is a string which represents a regular expression which is used by the predeclared function `split` when the second parameter is not given. The initial value of the variable is the string `"[ \t]+"`.

9.4.3 Pattern matching

The space `re` contains predeclared functions which are used for *pattern matching*. The pattern is described by *regular expressions* (*regex*) and actually a small program describing a string matching. The pattern has *default* syntax of ONIGURUMA package for Unicode. It is hard to describe formally the pattern syntax. Here is an incomplete strict description. For the full reference, please see OINGURUMA package documentation. The regular expressions have the following syntax:

```
Regex = Branch {"|" Branch}
```

The regex matches anything that matches one of the *branches*.

```
Branch = {Piece}
```

The branch matches the first *piece*, followed by the second piece, etc. If the pieces are omitted, the branch matches the null string.

```
Piece = Anchor | Unit

Unit = Atom
     | Unit Quantifier

Quantifier = Greedy
           | Reluctant
           | Possesive
```

```
Greedy = "?"                  // 0 or 1 times
       | "*"                  // 0 or more times
       | "+"                  // 1 or more times
       | Bound

Bound = "{" Min "," Max "}" // from Min to Max times
      | "{" Min "," "}"     // at least Min times
      | "{" "," Max "}"     // equivalent to {0, Max}
      | "{" Min "}"         // given number times

Reluctant = "??"
          | "*?"
          | "+?"
          | Bound "?"

Possesive : "?+"
          | "*+"
          | "++"

Min = <unsigned integer>

Max = <unsigned integer>
```

The *unit* followed by `*` matches a sequence of 0 or more matches of the unit. An unit followed by `+` matches a sequence of 1 or more matches of the unit. An unit followed by `?` matches a sequence of 0 or 1 matches of the unit.

There is a more general construction (a *bound*) for describing repetitions of an unit. An unit followed by a bound containing only one integer `Min` matches a sequence of exactly `Min` matches of the unit. An unit followed by a bound containing one integer `Min` and a comma matches a sequence of `Min` or more matches of the unit. An unit followed by a bound containing a comma and one integer `Max` matches at most `Max` repetitions of the unit. An unit followed by a bound containing two integers `Min` and `Max` matches a sequence of `Min` through `Max` (inclusive) matches of the unit.

The described above qualifiers are *greedy* ones. A gready qualifier first matches as much as possible and can back-track in a case of the whole regex matching failure to try shorter sequence. There are *reluctant* qualifiers too. They have additional suffix `?` and first they match as little as possible. The last type of the qualifiers is possesive. Such qualifiers have additional suffix `+` and behave like the corresponding greedy ones, but they do not back-track.

Examples:

```
`.?foo` // matches first "xfoo" in "xfooxxxxfoo"
`.*foo` // matches all "xfooxxxxfoo"
`.+foo` // matches all "xfooxxxxfoo"
`.{1,8}foo` // matches all "xfooxxxxfoo"
`.*?foo` // matches first "xfoo" in "xfooxxxxfoo"
`.+?foo` // Ditto
`.{1,8}?foo` // Ditto
`.*+foo` // fail to match in "xfooxxxxfoo"
`.++foo` // fail to match in "xfooxxxxfoo"
```

```
Atom =  Anchors
      | Character
      | CharacterType
      | CharacterProperty
      | CharacterClass
      | Group
      | BackReference
      | SubexpCall

 Character = "\t"     // horizontal tab (0x09)
           | "\v"     // vertical tab (0x0B)
           | "\n"     // newline tab (0x0A)
           | "\r"     // return (0x0D)
           | "\f"     // form feed (0x0C)
           | "\a"     // bell (0x07)
           | "\e"     // escape (0x1B)
           | "\" OctalCode // char with given octal code
           | "\x" HexCode  // char with given hexadecimal code
           | <any but special character \ ? * + ^ $ [ ( ) >
           | "\" <special character>

 OctalCode = <3 octal digits>

 HexCode = <2 heaxadecimal digits>

 CharacterType = '.'  // any character but newline
               | "\w" // Unicode Letter, Mark, Number, or
                      //   Connector_Punctuation
               | "\W" // opposite to the above
               | "\s" // Unicode Line_Separator,
                      //   Paragraph_Separator, or
                      //   Space_Separator
               | "\S" // opposite to the above
               | "\d" // Unicode decimal number
               | "\D" // opposite to the above
               | "\h" // hexadecimal digit char [0-9a-fA-F]
               | "\H" // opposite to the above

 CharacterProperty = "\p{" PropertyName "}"
                   | "\p{^" PropertyName "}"
                   | "\P{" PropertyName "}"

PropertyName = "Alnum" | "Alpha" | "Blank" | "Cntrl"
             | "Digit" | "Graph" | "Lower" | "Print"
             | "Punct" | "Space" | "Upper" | "XDigit"
             | "Word" | "ASCII"
             | "Any" | "Assigned" | "C" | "Cc" | "Cf"
             | "Cn" | "Co" | "Cs" | "L" | "Ll" | "Lm"
             | "Lo" | "Lt" | "Lu" | "M" | "Mc" | "Me"
             | "Mn" | "N" | "Nd" | "Nl" | "No" | "P"
             | "Pc" | "Pd" | "Pe" | "Pf" | "Pi" | "Po"
```

```
                        | "Ps" | "S" | "Sc" | "Sk" | "Sm" | "So"
                        | "Z" | "Zl" | "Zp" | "Zs" | "Arabic"
                        | "Armenian" | "Bengali" | "Bopomofo"
                        | "Braille" | "Buginese" |  "Buhid"
                        | "Canadian_Aboriginal" | "Cherokee"
                        | "Common" | "Coptic" | "Cypriot"
                        | "Cyrillic" | "Deseret" | "Devanagari"
                        | "Ethiopic" | "Georgian" |  "Glagolitic"
                        | "Gothic" | "Greek" | "Gujarati"
                        | "Gurmukhi" | "Han" | "Hangul" | "Hanunoo"
                        | "Hebrew" | "Hiragana" | "Inherited"
                        | "Kannada" | "Katakana" | "Kharoshthi"
                        | "Khmer" | "Lao" | "Latin" | "Limbu"
                        | "Linear_B" | "Malayalam" | "Mongolian"
                        | "Myanmar" | "New_Tai_Lue" | "Ogham"
                        | "Old_Italic" | "Old_Persian" | "Oriya"
                        | "Osmanya" | "Runic" | "Shavian" | "Sinhala"
                        | "Syloti_Nagri" | "Syriac" | "Tagalog"
                        | "Tagbanwa" | "Tai_Le" | "Tamil" | "Telugu"
                        | "Thaana" | "Thai" | "Tibetan" | "Tifinagh"
                        | "Ugaritic" | "Yi"

Anchors = "^"           // beginning of the line
        | "$"           // end of the line
        | "\b"          // word boundary
        | "\B"          // not word boundary
        | "\A"          // beginning of string
        | "\Z"          // end of string, or before newline
                        //   at the end
        | "\z"          // end of string
```

The atom can be a character. Some characters has a special meaning in regex (see comments in the character syntax). The rest characters match the same character in the matching string. To match a special character, use \ before the character. Some characters can be represented by a sequence starting with \ (see the syntax comments).

Examples:

```
‘\t‘        // matches "\\t"
‘\x65‘      // matches "e"
‘\p{Alpha}‘ // matches "a"
‘\w‘        // matches "a"
‘b$‘        // matches "b" in "b\na"
```

The atom can be an anchor. Matching anchors succeeds only if their positions correspond a specific place at the matching string (see comments in the anchor syntax).

Examples:

```
‘b$‘        // matches "b" in "b\na"
‘abc\Z‘     // matches "abc" in "abc"
‘abc\Z‘     // matches "abc" in "abc\n"
```

The atom which is a character type matches a specific class of character (see comments in the character type syntax).

The atom which is a character property matches a specific class of characters. For meaning `Alnum` - `ASCII`, please see the corresponding `BracketClass`. For meaning `C` - `Zs`, please see Unicode categories. For meaning `Armenian` - `Yi`, please see the Unicode scripts (alphabets). If the property contains `p` with `^` or `P`, the match succeeds when the matching character is not of the class.

Examples:

```
‘\p{Alpha}‘ // matches "a"
‘\p{ASCII}‘ // matches ";"

CharacterClass = "[" Intersections "]"
               | "[^" Intersections "]"

Intersections = Set
              | Intersections "&&" Set

Set = SetElement
    | Set SetElement

SetElement = ElementChar ["-" ElementChar]
           | "[:" BracketClass ":]"
           | "[:^" BracketClass ":]"
           | CharacterClass

ElementChar = Character
            | "\b"       // backspace 0x08

BracketClass = "alnum"   // Unicode letter, mark,
                         //   or decimal number
             | "alpha"   // Unicode letter or mark
             | "ascii"   // character in range 0 - 0x7f
             | "blank"   // Unicode space separator
                         //   or \t (0x09)
             | "ctrl"    // Unicode control, format,
                         //   unassigned, private use,
                         //   or surrogate
             | "digit"   // Unicode decimal number
             | "graph"   // not a space class and not an
                         //   Unicode control, unassigned,
                         //   or surrogate
             | "lower"   // Unicode lower case letter
             | "print"   // graph or space class
             | "punct"   // any Unicode punctuation
             | "space"   // any Unicode separator,
                         //   \t (0x09), \n (0x0A), \v (0x0B),
                         //   \f (0x0C), \r (0x0D),
                         //   or 0x85 (next line)
             | "upper"   // Unicode upper case letter
             | "xdigit"  // ascii 0-9, a-f, or a-f
```

```
                    | "word"    // Unicode letter, mark, decimal
                                //   number or punctuation connector
```

The atom can be a bracket expression which is a *list* of intersections of character sets separated by `&&` and enclosed in `[]`. If the character class contains `^` right after `[`, it matches any character which does match the corresponding character class without `^`. A set is a sequence of set elements.

The element given by a character denotes the character itself. An element given by two characters in the list separated by `-` is shorthand for the full *range* of characters between those two (inclusive) in the sequence of the unicode codes, e.g. `[0-9]` matches any decimal digit. Besides the usual character representation you can use here also `\b` which is a backspace representation.

The element given by a bracket class enclosed in `[[::]]` matches a character from this class (see comments in BracketClass). If character `^` is present right after `[[:`, the match succeeds if the character is not in this class.

The element can be given by a character class, in other words the character clases can be nested.

If you need to use `[`, `-`, or `]` as a normal character in a character class, you can use prefix `\` for this.

Examples:

```
‘[[:alpha:]]‘  // matches "a"
‘[[[:lower:]]&&[^a-x]]‘ // matches "y" or "z"
```

The atom can be a group, a regular expression enclosed in `()`. There are several types of groups:

```
Group = CapturedGroup
      | NonCapturedGroup
      | "(?#" <any characters but )> ")" // a comment
      | "(?" Options ")"
      | Context

Options =
        | Options Option

Option = "-" | "i" | "m" | "x"

CapturedGroup = "(" [Regex] ")"
              | "(?<" Name ">" [Regex] ")"

Name = <one or more word character>

NonCapturedGroup = "(?" Options ":" [Regex] ")"
                 | "(?>" [Regex] ")" /* Atomic group */

Context = "(?=" [Regex] ")" // look ahead
        | "(?!" [Regex] ")" // negative look ahead
        | "(?<=" [Regex] ")" // look behind
        | "(?<!" [Regex] ")" // negative look behind

BackReference = "\" Number    // back ref. by group
                              //   number
```

```
                             | "\k<" Number ">" // back ref. by group
                                                //   number
                             | "\k<-" Number ">" // back ref. by relative
                                                 //   group number
                             | "\k<" Name ">" // back ref. by group name
                             // back ref. by group name and nest level:
                             | "\k<" Name "+" | "-" Number ">"

              Number = <any integer >= 0>
```

Some groups are captured groups. It means that you can refer the substrings they match (see the back references) or get the start and the end positions of the matched substrings by calling the Dino regex match functions. A captured group may have a name which can be used in the back references or in the subexp calls.

You can place comments not containing `)` in regex betweeen `(?#` and `)`.

Options without a regex always matches. They just change how matching works. The option `i` switches on igoring the letter cases during the match. The pption `m` makes `.` to match a newline too. The option `x` switches on ignoring the white spaces as a character atom and permits to add comments starting with `#` and ending at the end of line. The character `-` after the corresponding `?` has an opposite effect, e.g. it makes a letter case important in matching again etc.

You can define the options in non captured groups. These options affect only this group. Another form of non-captured group is an atomic group. Once regex in an atomic group mathes something, the matching stays the same during back-tracking.

Examples:

```
              `(?i:ab)`     // matches "Ab"
              `(?x: a a a)` // matches "aaa"
              `(?>.*)c`     // can not match "abc"
```

The atom can be a context. A context match does not advance the current position in a matching string. A look ahead context succeeds if the corresponding regex matches a sub-string starting from the current position. A look behind context succeeds if the corresponding regex matches a sub-string finishing right before the current position. There are negative forms of the context atom. They succeed when the corresponding regex does not match.

Examples:

```
              `(?=bcd)bc`   // matches "bc" in "aabcd"
              `(?<=aa)bc`   // matches "bc" in "aabc"
```

The atom can be a back reference. It refers to the matched string of the corresponding captured group. The captured groups are counted by their left parantheses starting from one going from left to right. The negative number denotes relative order number, in other words, the order is taken starting from the back reference going from right to left. If the captured group has a name, its matched string can be referenced by its name. If several group has the same name, the name in the back reference corresponds to the last such group. You can add a nest level to the name. If the nest level is zero it is the same as named back reference without nested level. A back reference with non-zero nest level never matches.

Examples:

```
‘(a)\k<1>‘     // matches "aa"
‘(?<p>a)\k<p>‘ // Ditto
```

The Atom can be a subexp call:

```
SubexpCall = "\g<" Name ">"
```

The subexp call is actually another occurence of the group it refers to. But if the call is in the group it refers, it is a recursive description. Only left recursion is not permitted as this results in never ending recursion.

Examples:

```
‘(?<p>cd)\g<p>‘   // matches "cdcd"
‘(?<p>a|b\g<p>c)‘ // matches "a", "bac", "bbacc" etc
‘(?<p>a|b\g<p>c)‘ // wrong left recursion.
```

There are the following pattern matching functions in the space `re`:

- `match (regex, string)`. The function searches for matching the regular expression `regex` in the `string`. The both parameters should be strings after their implicit string conversions. The matching is made according to the standard POSIX 1003.2: The regular expression matches the substring starting earliest in the string. If the regular expression could match more than one substring starting at that point, it matches the longest. Subexpressions also match the longest possible substrings, subject to the constraint that the whole match be as long as possible, with subexpressions starting earlier in the regular expression taking priority over ones starting later. In other words, higher-level subexpressions take priority over their component subexpressions. Match lengths are measured in characters, not the collating elements. A null string is considered longer than no match at all.

  If there is no matching, the function returns the value `nil`. Otherwise, the function returns a new mutable vector of integers. The length of the vector is `2 * (N + 1)` where `N` is the number of the captured groups. The first two elements are the index of the first character of the substring corresponding to the whole regular expression and the index of the last character matched plus one. The subsequent two elements are the index of the first character of the substring corresponding to the first captured group in the regular expression and the index of the last character plus one, and so on. If there is no matching with a captured group, the corresponding vector elements will have negative values.

  Example: The program

  ```
  println (re.match (‘\n()(a)((a)(a))‘, "b\naaab"));
  ```

  outputs

  ```
  [1, 5, 2, 2, 2, 3, 3, 5, 3, 4, 4, 5]
  ```

- `gmatch (regex, string[, flag])`. The function searches for different occurrences of the regular expression `regex` in `string`. Both parameters should be strings after their implicit string conversion. The third parameter is optional. If it is present, it should be integer after an implicit integer conversion. If its value is nonzero, the substrings matched by regex can be overlapped. Otherwise, the substrings are never overlapped. If the parameter is absent, the function behaves as its value were zero. The function returns a new mutable vector of integers. The length of the vector is `2 * N` where `N` is number of the found occurrences. Pairs of the vector elements correspond to the occurrences. The first element of the pairs is an index of the first character of substring corresponding to all regular expression in the corresponding occurrences and the second element is an index of the last character plus one. If there is no one occurrence, the function returns `nil`. Example: The program

```
println (re.gmatch (‘aa‘, "aaaaa"));
println (re.gmatch (‘aa‘, "aaaaa", 1));
```

outputs

```
[0, 2, 2, 4]
[0, 2, 1, 3, 2, 4, 3, 5]
```

- `sub (regex, string, subst)`. The function searches for substrings matching the regular expression `regex` in `string`. All parameters should be string after an implicit string conversion. If there is no matching, the function returns the value `nil`. Otherwise, the function returns a new mutable vector of characters in which the first substring matched has been changed to the string `subst`. Within the replacement string `subst`, the sequence \`n`, where `n` is a digit from 1 to 9, may be used to indicate the text that matched the `n`'th captured group of the regex. The sequence \`0` represents the entire matched text, as does the character `&`.
- `gsub (regex, string, subst)`. The function is analogous to the function `sub` except for the function searches for all non-overlapping substrings matched with the regular expression and returns a new mutable vector of characters in which all matched substrings have been changed to the string `subst`.
- `split (string [, regex])`. The function splits `string` into non-overlapped substrings separated in the input string by strings matching the regular expression. All parameters should be strings after an implicit string conversion. If the second parameter is omitted the value of the predeclared variable `split_regex` is used instead of the second parameter value. In this case the function may generate the exception `invenvar` (corrupted value of a predeclared variable). The function returns a new mutable vector with the elements which are the separated substrings. If the regular expression is the null string, the function returns a new mutable vector with the elements which are strings each containing one character of string.

  Examples: The program

  ```
  println (re.split ("aaa bbb ccc       ddd"));
  ```

  outputs

  ```
  ["aaa", "bbb", "ccc", "ddd"]
  ```

  The program

  ```
  println (re.split ("abcdef", ‘‘));
  ```

  outputs

  ```
  ["a", "b", "c", "d", "e", "f"]
  ```

If the regular expression is incorrect, the functions generate the exception `invregex` with a message explaining the error.

9.5 Space `math`

The space contains mostly mathematical functions.

9.5.1 Mathematical functions

The following functions make an implicit arithmetic conversion of the parameters. After the conversions the parameters are expected to be of integer, long integer, or floating point type. The result is always a floating point number.

- `sqrt (x)`. The function returns the square root of `x`. The function generates the exception `edom` if `x` is negative.
- `exp (x)`. The function returns `e` (the base of the natural logarithm) raised to the power of `x`.
- `log (x)`. The function returns the natural logarithm of `x`. The function generates the exception `edom` if `x` is negative or may generate `erange` if the value is zero.
- `log10 (x)`. The function returns the decimal logarithm of `x`. The function generates the exception `edom` if `x` is negative or may generate `erange` if the value is zero.
- `pow (x, y)`. The function returns `x` raised to the power of `y`. The function generates exception `edom` if x is negative and y is not of integral value.
- `sin (x)`. The function returns the sine of `x`.
- `cos (x)`. The function returns the cosine of `x`.
- `atan2 (x, y)`. The function returns the arc tangent of the two variables `x` and `y`. It is similar to calculating the arc tangent of `y / x`, except that the signs of both arguments are used to determine the quadrant of the result.

9.5.2 Other space `math` functions

There are the following miscellaneous functions:

- `max (v1, v2, ...)`. The function searches for and returns the maximal value in all of its parameters. The parameters should be of integer, long integer, or floating point type after an implicit arithmetic conversion. So the function can return an integer, a long integer, or floating point number depending on the type of the first maximal value after the conversion.
- `min (v1, v2, ...)`. The function is analogous to the previous function, but searches for and returns the minimal value.
- `srand ([seed])`. The function sets the parameter value (after an implicit integer conversion) as a seed for a new sequence of pseudo-random integers to be returned by `rand`. These sequences are repeatable by calling `srand` with the same seed value. If the parameter is not given, the seed will be the result of calling function `time`.
- `rand ()`. The function returns a pseudo-random floating point value between 0 and 1. If the function `srand` was not called before, 1 will be used as the seed value.

9.6 Space `yaep`

This space contains declarations to work with Yet Another Earley Parser (YAEP). YAEP is a very powerful tool to implement language compilers, processors, or translators. The implementation of the Earley parser used in Dino has the following features:

- It is sufficiently fast and does not require much memory. This is the fastest implementation of the Earley parser which I know. The main design goal was to achieve a speed and memory requirements which are necessary to use it in prototyping compilers and language processors. It can parse 300,000 lines of C code per second on modern computers and allocates about 5MB memory for a 10,000 line C program.
- It makes a simple syntax directed translation, so an abstract tree is already the output of the Earley parser.
- It can parse input described by an ambiguous grammar. In this case the parse result can be an abstract tree or all possible abstract trees. Moreover, it produces the compact representation of all possible parse trees by using DAG instead of real trees. These features can be used to parse natural language sentences.
- It can make a syntax error recovery. Moreover its error recovery algorithms find an error recovery with a minimal number of ignored tokens. It permits an implemention of parsers with very good error recovery and reporting.
- It has a fast startup. There is practically no delay between processing of the grammar and the start of parsing.
- It has a flexible interface. The input grammar is given by a YACC-like description.
- It has a good debugging features. It can print huge amount of information about grammar, parsing, error recovery, translation. You can even get the result translation in a form for a graphic visualization program.

9.6.1 Exception classes of space `yaep`

The space `yaep` contains the class `invparser` which is a sub-class of `invcall`. The following sub-classes of the class `invparser` describe exceptions specific for the work with YAEP.

- `invgrammar`. This class describes an exception that the Earley parser got a bad grammar, e.g. without rules, with loops in rules, with nonterminals unachievable from the axiom, with nonterminals not deriving any terminal string etc.
- `invtoken`. This class describes an exception that the parser got an input token with unknown (undeclared) code.
- `pmemory`. This class describes an exception that there is not enough memory for the internal parser data.

9.6.2 Class parser

The space `yaep` has the predeclared final class `parser` which implements Earley parser. The following public functions and variables are declared in the class `parser`:

- `ambiguous_p`. This public variable stores information about the last parsing. A nonzero variable value means that during the last parsing on a given inputm the parser found that the grammar is ambiguous. The parser can find this even if you asked for only one parser tree (see the function `set_one_parse`).
- `set_grammar (descr, strict_p)`. This function tunes the parser to given grammar. The grammar is given by the string `descr`. A nonzero value of the parameter `strict_p` (after an implicit integer conversion) means more strict grammer checking. In this case, all nonterminals will be checked on their ability to derive a terminal string instead of only checking the axiom for this. The function can generate the exceptions `partype` (if the parameters have wrong types) or `invgrammar` if the description is a bad grammar. The function can also generate the exception `pmemory` if there is no memory for the internal parser data.

  The description is similiar to the *YACC* one. It has the following syntax:

  ```
  file : file terms [';']
       | file rule
       | terms [';']
       | rule

  terms : terms IDENTIFIER ['=' NUMBER]
        | TERM

  rule : IDENTIFIER ':' rhs [';']

  rhs : rhs '|' sequence [translation]
      | sequence [translation]

  sequence :
           | sequence IDENTIFIER
           | sequence C_CHARACTER_CONSTANT

  translation : '#'
              | '#' NUMBER
              | '#' '-'
              | '#' IDENTIFIER [NUMBER] '(' numbers ')'

  numbers :
          | numbers NUMBER
          | numbers '-'
  ```

  So the description consists of terminal declaration and rule sections.

  The terminal declaration section describes the names of terminals and their codes. The terminal code is optional. If it is omitted, the terminal code will the next free code starting with 256. You can declare a terminal several times (the single condition is that its code should be the same).

A character constant present in the rules is a terminal described by default. Its code is always the ASCII code of the character constant.

The rules syntax is the same as *YACC* rule syntax. The single difference is an optional translation construction starting with `#` right after each alternative. The translation part could be a single number which means that the translation of the alternative will be the translation of the symbol with the given number (symbol numbers in the alternative start with 0). Or the translation can be empty or '`-`' which designates the value of the variable `nil_anode`. Or the translation can be an abstract node with the given name, optional cost, and with the fields whose values are the translations of the alternative symbols with numbers given in parentheses after the abstract node name. You can use '`-`' in an abstract node to show that the empty node should be used in this place. If the cost is absent it is believed to be 1. The cost of the terminal, error node, and empty node is always zero.

There is a reserved terminal `error` which marks the start point of an error recovery. The translation of the terminal is the value of the variable `error_anode`.

- `set_debug (level)`. This function sets up the level of debugging information output to `stderr`. The higher the level, the more information is output. The default value is 0 (no output). The debugging information includes statistics, the result translation tree, the grammar, parser sets, parser sets with all situations, situations with contexts. The function returns the previously set up debug level. Setting up a negative debug level results in output of the translation for the utility **dot** of the graphic visualization package **graphviz**. The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `set_one_parse (flag)`. This function sets up a flag whose nonzero value means building only one translation tree (without any alternative nodes). For an unambiguous grammar the flag does not affect the result. The function returns the previously set up flag value. The default value of the flag is 1. The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `set_lookahead (flag)`. This function sets up a flag of usage of a look ahead in the parser work. The usage of the lookahead gives the best results with the point of view of the space and speed. The default value is 1 (the lookahead usage). The function returns the previously set up flag. No usage of the lookahead is useful sometimes to get more understandable debug output of the parser work (see the function `set_debug`). The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `set_cost (flag)`. This function sets up building the only translation tree (trees if we set up one_parse_flag to 0) with minimal cost. For an unambiguous grammar the flag does not affect the result. The default value is 0. The function returns the previously set up flag value. The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `set_recovery (flag)`. This function sets up a flag whose nonzero value means making error recovery if a syntax error occurred. Otherwise, a syntax error results in finishing parsing (although the syntax error function passed to `parse` still be called once). The function returns the previously set up flag value. The default value of the flag is 1. The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `set_recovery_match (n_toks)`. This function sets up an internal parser parameter meaning how much subsequent tokens should be successfully shifted to finish the error recovery. The default value is 3. The function returns the previously set up value. The parameter should be an integer after an implicit integer conversion. The function will generate the exception `partype` if it is not true.

- `parse (tokens, error_func)`. This function is the major function of the class. It makes the translation according to the previously set up grammar of input given by the parameter `tokens` whose value should be an array of objects of predeclared class `token` or of its subtype. If the parser recognizes a syntax error it calls the function given through parameter `error_func` with six parameters:

  - an index of the token (in the array `tokens`) on which the syntax error occured.
  - the error token itself. It may be `nil` for end of file.
  - an index of the first token (in the array `tokens`) ignored due to error recovery.
  - the first ignored token itself. It may be `nil` for end of file.
  - an index of the first token (in the array `tokens`) which is not ignored after the error recovery.
  - the first not ignored token itself. It may be `nil` for end of file.

  If the parser works with switched off error recovery (see the function `set_recovery`, the third and fifth parameters will be negative and forth and sixth parameter will be `nil`.

  The function returns an object of the predeclared class `anode` which is the root of the abtsract tree representing the translation of the parser input. The function returns `nil` only if a syntax error was occurred and the error recovery was switched off. The function can generate the exception `partype` if the parameter types are wrong or the exception `invtoken_decl` if any of the input tokens have a wrong code. The function also can generate teh exception `pmemory` if there is no memory for the internal parser data.

The call of the class `parser` itself can generate the exception `pmemory` if there is no memory for the internal parser data.

9.6.3 Class `token`

The space `yaep` has a predeclared class `token`. Objects of this class should be the input of the Earley parser (see the function `parse` in the class `parser`). The result abstract tree representing the translation will have input tokens as leaves. The class `token` has one public variable `code` whose value should be the code of the corresponding terminal described in the grammar. You could extend the class description e.g. by adding variables whose values could be attributes of the token (e.g. a source line number, the name of an identifier, or the value for a number).

Class `anode` The space `yaep` has a predeclared class `anode` whose objects are nodes of the abtract tree representing the translation (see teh function `parse` of class `parser`). Objects of this class are generated by Earley parser. The class has two public variables `name` whose value is a string representing a name of the abstract node as it given in the grammar and `transl` whose value is an array with abstract node fields as the array elements. There are a few node types which have special meaning:

- A terminal node which has the reserved name `$term`. The value of the public variable `transl` for this node is an object of the class `token` representing the corresponding input token which was an element of the array passed as a parameter of the function `parse`.

- An error node which has the reserved name `$error`. This node exists in one exemplar (see description of the variable `error_anode`) and represents the translation of the reserved grammar symbol `error`. The value in the public variable `transl` will be `nil` in this case.
- An empty node which has the reserved name `$nil`. This node also exists in one exemplar (see description of the variable `nil_anode`) and represents the translation of a grammar symbol for which we did not describe a translation. For example, in a grammar rule an abstract node refers for the translation of a nonterminal for which we do not produce a translation. The value in the public variable of such class object will be `nil` in this case.
- An alternative node which has the reserved name `$alt`. It represents all possible alternatives in the translation of the grammar nonterminal. The value of the public variable `transl` will be an array with elements whose values are objects of the class `anode` which represent all possible translations. Such nodes can be generated by the parser only if the grammar is ambiguous and we did not ask it to produce only one translation.

9.6.4 Variables `nil_anode` and `error_anode`

There is only one instance of `anode` which represents empty (nil) nodes. The same is true for the error nodes. The final variables `nil_anode` and `error_anode` correspondingly refer to these nodes.

9.6.5 Example of Earley parser usage.

Let us write a program which transforms an expression into the postfix polish form. Please, read the program comments to understand what the code does. The program should output string `"abcda*+*+"` which is the postfix polish form of input string `"a+b*(c+d*a)"`.

```
expose yaep.*;
// The following is the expression grammar:
var grammar = "E : E '+' T   # plus (0 2)\n\
                 | T         # 0\n\
                 | error     # 0\n\
               T : T '*' F   # mult (0 2)\n\
                 | F         # 0\n\
               F : 'a'       # 0\n\
                 | 'b'       # 0\n\
                 | 'c'       # 0\n\
                 | 'd'       # 0\n\
                 | '(' E ')' # 1";
// Create the parser and set up the grammar.
var p = parser ();
p.set_grammar (grammar, 1);

// Add attribute repr to the token:
class our_token (code) { use token former code; var repr; }
// The following code forms input tokens from the string:
var str = "a+b*(c+d*a)";
var i, inp = [#str : nil];
for (i = 0; i < #str; i++) {
  inp [i] = our_token (str[i] + 0);
  inp [i].repr = str[i];
}
// The following function outputs messages about the syntax errors
// and the syntax error recovery:
fun error (err_start, err_tok,
            start_ignored_num, start_ignored_tok_attr,
            start_recovered_num, start_recovered_tok) {
  put ("syntax error on token #", err_start,
       " (" @ err_tok.code @ ")");
  putln (" -- ignore ", start_recovered_num - start_ignored_num,
         " tokens starting with token #", start_ignored_num);
}

var root = p.parse (inp, error); // parse

// Output the translation in the polish inverse form
fun pr (r) {
  var i, n = r.name;

  if (n == "$term")
    put (r.transl.repr);
  else if (n == "mult" || n == "plus") {
    for (i = 0; i < #r.transl; i++)
      pr (r.transl [i]);
    put (n == "mult" ? "*" : "+");
  }
  else if (n != "$error") {
    putln ("internal error");
    exit (1);
  }
}

pr (root);
putln ();
```

10 Appendix A. The Dino syntax

```
Expr = Expr "?"  Expr ":" Expr
     | Expr "||"  Expr
     | Expr "&&"  Expr
     | Expr in  Expr
     | Expr "|"  Expr
     | Expr "^"  Expr
     | Expr "&"  Expr
     | Expr "=="  Expr
     | Expr "!="  Expr
     | Expr "==="  Expr
     | Expr "!=="  Expr
     | Expr "<"  Expr
```

```
          | Expr ">"  Expr
          | Expr "<="  Expr
          | Expr ">="  Expr
          | Expr "<<"  Expr
          | Expr ">>"  Expr
          | Expr ">>>"  Expr
          | Expr "@"  Expr
          | Expr "+"  Expr
          | Expr "-"  Expr
          | Expr "*"  Expr
          | Expr "/"  Expr
          | Expr "%"  Expr
          | "!"  Expr
          | "+"  Expr
          | "-"  Expr
          | "~"  Expr
          | "#"  Expr
          | final  Expr
          | new  Expr
          | ".+"  Expr
          | ".*"  Expr
          | ".&"  Expr
          | ".^"  Expr
          | ".|"  Expr
          | Designator
          | INTEGER
          | FLOATINGPOINTNUMBER
          | CHARACTER
          | nil
          | "(" Expr ")"
          | Call
          | "["  ElistPartsList "]"
          | tab "["  ElistPartsList "]"
          | STRING
          | char
          | int
          | long
          | float
          | hide
          | hideblock
          | vec
          | tab
          | fun
          | fiber
          | class
          | obj
          | thread
          | type
          | this
          | char "(" Expr ")"
          | int "(" Expr ")"
```

```
     | float "(" Expr ")"
     | vec "(" Expr ["," Expr] ")"
     | tab "(" Expr ")"
     | type "(" Expr ")"
     | AnonHeader FormalParameters Block
     | try "(" ExecutiveStmt [ ","  ExceptClassList] ")"
     | "_"
     | "..."

Designator = DesignatorOrCall "["  Expr "]"
           | DesignatorOrCall ActualParameters
           | DesignatorOrCall "."  IDENT
           | IDENT

ElistPartsList = [ Expr [ ":" Expr ] {"," Expr [ ":" Expr ] } ]

DesignatorOrCall = Designator
                 | Call

Call = Designator ActualParameters

ActualParameters = "(" [ Expr { "," Expr } ] ")"

AnonHeader = [Qualifiers] FuncFiberClass

Stmt = ExecutiveStmt
     | Declaration

Assign = "="
       | "*="
       | "/="
       | "%="
       | "+="
       | "-="
       | "@="
       | "<<="
       | ">>="
       | ">>>="
       | "&="
       | "^="
       | "|="

Pattern = Expr

ExecutiveStmt = ";"
              | Expr ";"
              | Designator Assign  Expr ";"
              | Designator ("++" | "--")  ";"
              | ("++" | "--")  Designator ";"
              | if  "(" Expr ")" Stmt [ else Stmt ]
              | for  "(" Stmt ForGuardExpr ";"  Stmt ")" Stmt
              | for  "(" Designator in Expr ")" Stmt
              | (pmatch | rmatch) "(" Expr ")" "{" CaseList "}"
              | break ";"
              | continue ";"
              | return  [ Expr ] ";"
              | throw  Expr ";"
              | wait  "(" Expr ")" Stmt
              | BlockStmt
              | TryBlockStmt
              | C_CODE

ForGuardExpr = [Expr]

CaseList = { case Pattern [CaseCond] ":" StmtList }
CaseCond = if Expr

BlockStmt = Block

TryBlockStmt = try Block { Catch }

Catch = catch  "(" ExceptClassList ")" Block

ExceptClassList = Expr { "," Expr }

Declaration = VarDeclarations
            | FriendClause
            | ExternDeclarations
            | FuncClassDeclaration
            | SingletonObject
            | ForwardDeclaration
            | ExposeClause
            | UseClause
            | IncludeDeclaration

VarDeclarations =  [pub | priv] (val | var)  VarList ";"

VarList = Var { "," Var }

Var = IDENT | pattern "="  Expr

ExternDeclarations = [pub | priv] extern ExternItems ";"

FuncClassDeclaration = Header FormalParameters Hint Block

Hint = [ "!" IDENT ]

FriendClause = friend IDENT { "," IDENT } ";"

IncludeDeclaration = include ["+"] STRING ";"

ExternItems = ExternItem { "," ExternItem }
```

```
ExternItem = IDENT
           | IDENT  "(" ")"

Header = [Qualifiers] FuncFiberClass IDENT

Qualifiers = pub | priv | final
           | pub final | priv final
           | final pub | final priv

FuncFiberClass = fun
               | fiber
               | class

FormalParameters =
                 | "(" [ ParList ] ")"
                 | "(" ParList "," "..." ")"
                 | "(" "..." ")"

ParList = Par { "," Par}

Par = [pub | priv] [val | var] IDENT [ "=" Expr]

ExposeClause = expose ExposeItem { "," ExposeItem }

ExposeItem = QualIdent ["(" IDENT ")"] | QaulIdent ".*"

QualIdent = IDENT {"." IDENT}

UseClause = use IDENT { UseItemClause }

UseItemClause = [former | later] UseItem { "," UseItem }

Item = IDENT [ "(" IDENT ")"]

Block = "{"  StmtList "}"

StmtList = { Stmt }

Program = StmtList
```

11 Appendix B. Implementation

```
Dino(1)                          User Manuals                         Dino(1)



NAME
       dino  -  the  interpreter of the programming language Dino (development
       version)
```

```
SYNOPSIS
       dino [ -h --help -s --statistics -t --trace -m size -Idirname -Lpath -O
       --check --save-temps -d --dump -p --profile] (-c program | program-file
       | -i dump-file ) dino-program-arguments

DESCRIPTION
       dino interprets a program in the Dino programming language.   The  proâĂŘ
       gram file (and include files) must have the suffix .d

       The  program  encodings  are  defined by the current encoding.  You can
       change  the  current  input  encoding  defining  environment   variable
       DINO_ENCODING  or  calling  DINO function set_encoding in case of REPL.
       Each program file can define own encoding putting  -*- coding:  <encodâĂŘ
       ing-name> -*-  anywhere on the first two lines of the file.

       The  description  of  Dino language is in the report of the Programming
       Language Dino.

OPTIONS
       Calling the interpreter without arguments is to call REPL (an  interacâĂŘ
       tive  environment  for  execution  of  Dino statements and printing the
       results of the statement execution).  The options which the Dino interâĂŘ
       preter recognizes are:

       -c program
              Execute  the Dino program given on the command line as the arguâĂŘ
              ment.

       -h, --help
              Output information about Dino command line options.

       -m number
              Determine the size of the heap chunks used by  the  Dino  interâĂŘ
              preter.   The  size can be given in bytes (e.g. 32000), in kiloâĂŘ
              bytes (e.g. 64k), or in megabytes (e.g. 1m).  The  default  size
              is  1  Megabyte.   Initially,  the  Dino interpreter creates one
              chunk.  It allocates one additional chunk (as rule of  the  same
              size)  whenever there is no additional memory after garbage colâĂŘ
              lection.

       -s, --statistics
              Output some statistics of interpreter work into stderr.  StatisâĂŘ
              tics  contain  the maximal heap size, number of heap chunks, and
              number of collisions in hash  tables  which  are  used  for  the
              implementation of Dino tables.

       -t, --trace
              Output  information  about call stack of the program into stderr
              for unprocessed exception or calling exit function.  This output
              sometimes can be different from what people would expect because
              of tail call elimination.

       -Idirname
              Define the  directory  in  which  Dino  include  files  will  be
              searched  for.  The order of searching in directories given with
              this option is the same as the one on the command line.

       -Ldirname
              Define  where  to  search  for  external  libraries  (if  shared
              libraries  are  implemented  on  the  system.   This is true for
              Linux, Solaris, Irix, OSF, and Windows) in which the Dino exterâĂŘ
              nal  variables and functions will be searched for.  The order of
              searching in libraries given with this option is the same as one
              on the command line.

       -O     Do type inference and byte code specialization.  It can speed up
              the program execution, especially when JIT is used.

       -d, --dump
              Output Dino byte code to the standard output.

       -i     Read Dino byte code from a given file and execute it.   You  can
              use this option instead of providing Dino program file.

       --check
              Check non run-time errors without program execution.  The option
              also checks all C-code fragments in the Dino program.

       --save-temps
              If a Dino function processed by JIT compiler,  the  generated  C
              source  code  and  PIC  object code can be saved after finishing
              Dino interpreter in directory /tmp with this option.   The  file
              names  will  contain identifier number of the corresponding Dino
              interpreter process.

       -p, --profile
              Output profile information  into  stderr.   Profile  information
              contains  the  number of calls and execution times of all called
              functions and classes.

FILES
       file.d
              a Dino program file
       libdino.so
              a Dino shared library on some Unix systems.
       mpi.d
              the DINO file implementing multiple precision arithmetic.
       mpi.so
              the DINO shared library used for implementing MPI on  some  Unix
              systems.
       ieee.d
```

```
              the  Dino  file implementing IEEE standard floating point arithâĂŘ
              metic.
       ieee.so
              the Dino shared library used for implementing IEEE on some  Unix
              systems.
       ipcerr.d
              the  Dino  file  definining  exceptions of ipc/network software.
              This file is used by socket.d.
       ipcerr.so
              the Dino shared library used for  implementing  IPCERR  on  some
              Unix systems.
       socket.d
              the Dino file implementing work with sockets.
       socket.so
              the  Dino  shared  library  used for implementing SOCKET on some
              Unix systems.
       Dino interpreter creates temporary C and object files  for  Dino  funcâĂŘ
       tions processed by JIT compiler in directory /tmp.

ENVIRONMENT
       There  are the following environment variables which affect Dino behavâĂŘ
       ior:

       DINO_HOME
              If not null, it defines the places of the dino shared  libraries
              (such a library may be only on some Unix systems including Linux
              and  Solaris),  include  files,  and  dino   standard   external
              libraries.   The  places  are defined as the subdirectory lib in
              directory given by the environment variable value.   You  should
              define  the variable value on Windows if you installed the files
              in a directory other than C:\dino\lib

       DINO_PATH
              If not null, it defines the places of dino  include-files.   The
              value  of the variable has more priority than DINO_HOME but less
              priority than values given through -I options.

       DINO_LIB
              If not null, it defines places of dino shared library,  if  any.
              The   value  of  variable  has  more  priority  than  DINO_HOME.
              DINO_EXTERN_LIBS.

       DINO_EXTERN_LIBS
              If not null,  it  defines  paths  of  additional  Dino  external
              libraries.   The  libraries should be separated by ":" (on Unix)
              or ";" (on Windows).  The value has less  priority  than  values
              given in -L options.

       DINO_ENCODING
              If  not  null,  it defines the current encoding of dino program.
              By defaull the current encoding is UTF-8  encoding.   Use  iconv
              utility to get possible encoding names.

DIAGNOSTICS
       DINO diagnostics are self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@gcc.gnu.org

BUGS
       Please report bugs to https://github.com/dino-lang/dino/issues.



Dino                              2 Apr 2016                           Dino(1)
```