The Programming Language DINO

Vladimir Makarov, vmakarov@users.sourceforge.net. Edited by Serhei Makarov. Jan 25, 2006

This document describes the programming language DINO.

Contents

```
History
==========
  Apr 29, 2001.  Documentation of all objects corresponding to Earley's
                 parser was added.
```

```
May 05, 2001.  Semantics of private declarations are changed slightly.
               Earlier, they were accessible only by a separate
               identifier.  Now they are accessible inside their
               scope.  In other words, they can be accessible by ‘.’
               or ‘->’ inside the scope.

               Added profile option -p.

               Added IEEE standard floating point arithmetic.

               Declaration block friends.

Jun 22, 2001.  Function rcount is added.

Jul 19, 2001.  Functions getf and fgetf have been added.

Jun 23, 2001.  Function gmatch is added, rcount is removed.

Jun 25, 2001.  Operators char, int, float are added.

Sep 25, 2001.  Vector conversion with format.  New function rev.
               Del, ins, and insv now return the vector.

Oct 27, 2001.  New predefined variable version.  Additional
               parameter for getf and fgetf.

Oct 31, 2001.  Added special method destroy.

Nov 22, 2001.  New functions sput, sputln, sprint, sprintln added.

Dec 13, 2001.  New variables nil_anode and error_anode and new
               function set_cost.

Mar 15, 2002.  Function parse may return nil.

Mar 18, 2002.  New function set_lookahead.

Apr 30, 2002.  Calling destroy by finishing the program.

Dec 8, 2002.   Equality of instance, classes and functions means
               the same context.
Dec 14, 2003.  Making table key is immutable in assignment.

Jan 29, 2004.  New exception invfmt. New functions putf, fputf, sputf.

Jan 4,  2006.  New function trans.  Make function rev returning a new
               array.

Jan 15, 2006.  Removing depricated -> and dereference '*' operations
```

```
Jan 16, 2006.  New swap operation.

Jan 25, 2006.  Add sync statement in wait-stmt
```

1 Introduction

DINO is a high level dynamic-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, TCL, and Python. 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. Dino has possibilities of parallelism description and exception handling. Dino is an extensible language with the possibility of dynamic loading of libraries written on other languages. 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 description of the movement of dinosaurs in a project. It has been considerably redesigned and was 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. 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 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 blanks in strings). 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.

```
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"
```

   Examples:

```
line  line2  next_line  NextLine
```

2. *Numbers* are (unsigned) decimal integer or floating point numbers. Numbers start with a digit. Floating point numbers are distinguished by the presence of decimal point `.` or an exponent in the number.

```
Number = Integer | FloatingPointNumber

Integer = Digit {Digit}

FloatingPointNumber = Digit {Digit} "." { Digit } [Exponent]
                    | Digit {Digit} [Exponent]

Exponent = ("e" | "E") [ "+" | "-" ] Digit { Digit }
```

   Examples:

```
10
100.
1e2
100.0E+0
```

3. A Dino *character constant* denotes an ASCII character. The following sequences starting with the 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` - ASCII character with given octal code

- `\char` - ASCII 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 ASCII 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 sequence of ASCII characters enclosed in double quotes. There are the same sequences of ASCII characters with special meaning as in a character constant. To denote a double quote mark use 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"
```

5. The remaining essential 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" | "catch" | "char" | "class" | "continue"
        | "else" | "ext" | "extern"
        | "final" | "float" | "for" | "friend" | "func"
        | "hide" | "hideblock" | "if" | "in" | "int"
        | "new" | "nil" | "public" | "private" | "return"
        | "table" | "thread" | "throw" | "try" | "type"
        | "var" | "vector" | "wait"
```

6. *Comments* are considered analogous to blanks on the syntax level of the program. There are two types of comments. The first type is an arbitrary character sequence starting with /* and finishing with */. The second type of 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. 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, or a class. The identifier is then used to refer to the associated declaration (more correctly with the declaration instance).

```
Declaration = VarDeclarations
            | AccessClause
            | ExternDeclarations
            | FuncClassExtDeclaration
            | IncludeDeclaration
```

The scope of a declaration is textually from the start (not from the point of declaration!) to the end of the block to which the declaration belongs and hence to which the declaration is local. It excludes the scopes of declarations with the same identifier which are in nested blocks. In a block, a maximum of one declaration of the same identifier is possible.

It is important to understand the notion of instantiation of the declaration. This notion reflects program execution, not the static structure of 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 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 (parallel execution branches) 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 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;
      func r () {return j;}
      f = r;
    }
putln (f ());
```

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 *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 public, private, or as friend:

```
AccessClause = (public | private | friend) AccessList ";"

AccessList = IDENT { "," IDENT }
```

Examples:

```
public param1, param2;
private call_count;
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. If the latter failed, the declaration is searched in the covering block instance of the current block instance, and so on.
- 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. If the designator is a class object, its context is a class object, and the search failed, the search is continued in the covering class object etc. The exception `accessop` occurs if the declaration is not found with such identifier, or the declaration is private and the construction is not in the declaration scope and not inside a friend of the declaration scope.

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

```
anode      argv         atan2
chdir      chgmod       chomod     chumod
clock      close        cmpv       context
cos        curr_thread
```

```
del
eltype        env           error_anode   errors
except        excepts       exit          exp
fatime        fctime        fget          fgetf
fgetln        fgmode        fgn           file
flush         fmtime        fomode        fprint
fprintln      fput          fputln        fscan
fscanln       fsize         ftype         fumode
fun
gc            get           getcwd        getegn
geteun        getgn         getgroups     getf
getln         getpid        getun         gmatch
gsub
ins           inside        insv          invaccesses
invcalls      invexterns    invindexes    invkeys
invops        invparsers    invregexps    isatty
keys
log           log10
main_thread   match         max           min
mkdir
nil_anode
open
parser        pclose        popen         pow
print         println       put           putln
rand          readdir       remove        rename
rev           rmdir
scan          scanln        seek          signals
sin           sort          split         split_regex
sprint        sprintln      sput          sputln
sqrt          srand         stderr        stdin
stdout        strtime       sub           subv
syserrors     system        systemcalls
tell          time          time_format   token
tolower       toupper       trans
version
```

The following identifiers are predeclared in the class *except* mentioned above.

```
error
```

The following identifiers are predeclared in the class *error* mentioned above.

```
deadlock
invaccess     invcall       invenv        invindex
invkey        invop
signal        syncwait
```

The following identifiers are predeclared in the class *signal* mentioned above.

```
sigabrt       sigfpe        sigill        sigint
sigsegv       sigterm
```

The following identifiers are predeclared in the class *invop* mentioned above.

```
optype         opvalue
```

The following identifiers are predeclared in the class *invindex* mentioned above.

```
indexop        indextype      indexvalue
```

The following identifiers are predeclared in the class *invkey* mentioned above.

```
keyop          keyvalue
```

The following identifiers are predeclared in the class *invcall* mentioned above.

```
callop
eof
internal       invenvar       invextern      invfmt
invinput       invparser      invregexp      invresult
parnumber      partype
syncthreadcall syserror       systemcall
```

The following identifiers are predeclared in the class *syserror* mentioned above.

```
eaccess        eagain         ebadf          ebusy
echild         edeadlk        edom           eexist
efault         efbig          eintr          einval
eio            eisdir         emfile         emlink
enametoolong   enfile         enodev         enoent
enoexec        enolck         enomem         enospc
enosys         enotdir        enotempty      enotty
enxio          eperm          epipe          erange
erofs          espipe         esrch          exdev
```

The following identifiers are predeclared in the class *systemcall* mentioned above.

```
noshell
systemfail
```

The following identifiers are predeclared in the class *invparser* mentioned above.

```
invgrammar     invtoken
pmemory
```

The following identifiers are predeclared in the class *invregexp* mentioned above.

```
badpat
ebrack         ectype         eend           eescape
eparen         erange         esize          espace
esubreg
```

The following identifiers are predeclared in the class *invextern* mentioned above.

```
libclose
noextern       noexternsupp
```

The following identifiers are predeclared in the class *invaccess* mentioned above.

```
accessop       accessvalue
immutable
```

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. Assigning of the initial value to the variable instance is made after execution of the previous statements of the block. By default the initial value of variables is the special value **nil**. The value of the variable can not be changed after its initialization if its declaration contains the keyword **final**.

```
VarDeclarations = var  VarParList ";"

VarParList = VarPar { "," VarPar }

VarPar = [final] IDENT [ "="  Expr]
```

Examples:

```
var i = 0, j, k;
var final constant = 10, final nil_constant, l;
```

4.2 External Declarations

Dino permits to use functions written in other languages, e.g. C. The functions should have special prototypes and must have to access to the DINO standard procedural interface (SPI). Dino can also have access to variables of a special type declared in the source code in another language. The details of the implementation of such features and the DINO SPI are not described here (some details are given in appendix B). As rule, the external functions and variables will be implemented as dynamically loaded libraries. This is the 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.

```
ExternDeclarations = extern  ExternItem { "," ExternItem } ";"

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

Examples:

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

4.3 Functions, Classes, Extensions

A function/class declaration consists of a function/class header and a function/class block (body). The header specifies the function identifier and *formal parameters*. 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 absent or is not executed, the function returns **nil** by default. A class call returns an object of the class which can be considered as a block instance of the class body. The return-statement for classes must be without a result. *Thread-functions* are analogous to general functions. The difference is in that a new execution *thread* is created during the thread-function call, the return-statement inside thread-function must be without an expression, and the thread-function returns the corresponding (execution) thread. The execution thread finishes when the corresponding thread block finishes. Execution threads are executed parallelly. 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 *actual parameters* during a call of the function/class. The function can be called with any number of *actual parameters*. If the number of actual parameters is less than the formal parameters number, the remaining formal parameters are initialized by the special value **nil**. Otherwise if the number of actual parameters is more than the number of formal parameters, the remaining actual parameter values are ignored. In order to process all actual parameters, you should place `...` at the end of the list of formal parameter declarations. This means 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 less or equal to the number of formal parameters (not taking the implicit parameter `args` into account), the value of `args` will be the empty vector. The formal parameter can be initialized by a default value in a way analogous to variable initialization. The initialization is made only when the corresponding actual parameter value is **nil**.

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 if it is declared as public. You should be remember that although the function may have parameters and return a value, the garbage collector (or finishing the program) ignores the result value and does not pass actual parameters. The single exception when the function `destroy` is not called by finishing the program is the case when memory can not be allocated more. So the values of the parameters will be **nil** if the function is called by the garbage collector (or finishing the program). You may prevent removing the corresponding object 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 increase of the heap.

Instead of inheritance usually used in object-oriented languages, Dino supports *extension*. This feature permits to modify function/class behaviour. All code inside an extension body is inserted at the end of body of the function/class declared with the same identifier in the same block in the same order as the extensions are placed in the block. A function/class declared as *final* can not be extended.

```
FuncClassExtDeclaration = Header Block

Header = [final] FuncThreadClass IDENT FormalParameters
       | ext IDENT

FuncThreadClass = func
                | thread
```

```
                    | class

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

Examples:

The following is a parameterless class header:

```
class stack ()
```

The following is a class header with an initialization:

```
          class stack (max_height = var a = 1, b = 2;

putln (a, " ", b);
a<=>b;
putln (a, " ", b);

var ar = [1, 2, 3];
println (ar);
ar[0]<=>ar[2];
println (ar);

class s (i) {}
var c1 = s (0), c2 = s (3);
putln (c1.i, ' ', c2.i);
c1.i<=>c2.i;
putln (c1.i, ' ', c2.i);

var t = {"s" : 1, "t" : 2};
putln (t{"s"}, ' ', t{"t"});
t{"s"}<=>t{"t"};
putln (t{"s"}, ' ', t{"t"});
100)
```

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

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

The following example illustrates the usage of extensions:

```
class point (x = 0, y = 0) {
}
ext point {
  class circle (radius = 1) {
```

```
    func square () {return 3.14 * radius * radius;}
  }
}
ext point {
  ext circle {
    class ellipse (width) {
      func square () {
        ...
      }
    }
}
```

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

```
var objs_number = 0;
class obj () {
  private n, destroy;
  var n = objs_number;
  objs_number++;
  func destroy () {objs_number--; objs_number--;}
}
```

The following example illustrates threads:

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

    wait (!empty);
    res = b [first];
    first = (first + 1) % length;
    wait (1) empty = first == free;
    return res;
  }
  func produce (val) {
    wait (empty || free != first);
    b [free] = val;
    free = (free + 1) % length;
    wait (1) empty = 0;
  }
}

thread consumer (buffer) {
    func produce (val) {
      buffer.produce (val);
      put ("produce: ");
      println (val);
    }
    produce (10);
```

```
    produce (10.5);
    produce ("string");
    produce ('c');
    produce (nil);
}

thread producer (buffer) {
  var val;

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

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

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. Dino is a dynamic-typed language. This means that a variable can store any Dino value.

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, 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 or keys) 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 {"key0" : 10, "key1" : 20}
final 5
final ['a', 'b', 'c']
final "abc"
final {"key0" : 10, "key1" : 20}
```

Dino has the following types of values:

- the special value *nil*. This is the default value of all variables when a block starts. The value is represented by the keyword **nil**.

  ```
  Expr = nil
  ```

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

  ```
  Expr = CHARACTER
  ```

- *integer*. For its representation see `Integer` in the section *Vocabulary and Representation*. It is always stored as a 32-bit integer value.

  ```
  Expr = INTEGER
  ```

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

  ```
  Expr = FLOATINGPOINTNUMBER
  ```

- *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, exception `optype` is generated. If the repetition value is negative or zero, the element value will be absent in the vector. Elements of 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", {}]
[]
```

- *table*. This is a structured shared type value. A table value is represented by a list of key values (expression values) in figure parentheses { 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 it is not true in a table constructor, 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 = "{"  ElistPartsList "}"
```

  Examples:

```
{'a', 'b', 10:[10]}
{'a' : nil, 'b' : nil, 10 : [10]}
{[10, 'a', {10}] : 10, [10] : {20:20}}
{}
```

- *function*. Its value is represented by the function designator. It is important to remember that the function is bound to a context.
- *thread-function*. Its value is represented by the thread-function designator. It is important to remember that the thread-function is bound to a context.
- *class*. Its value is represented by the class designator. It is important to remember that the class is bound to a context.
- *block instance*. There is no Dino representation of such values.
- *thread*. There is no literal Dino representation of such values. A thread value is generated by calling a thread-function.
- *object*(class instance). This is a structured shared type value. There is no literal Dino representation of such values. Objects are generated by calling classes.
- *hide value*. A hide value can not be generated by a Dino code. They are generated by external functions.
- *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 q hideblock value has no such constraint. Also a hideblock is of shared type.
- *type*. The values of such types are returned by th special operator `type (expression)`.

```
Expr = char
     | int
     | float
```

```
| hide
| hideblock
| vector
| table
| func
| thread
| class
| func "(" ")"
| thread "(" ")"
| class "(" ")"
| type
```

There are the following type values:

- type of **nil**. There is no value representing type of **nil**. So use the construction `type (nil)` to get it.
- type of characters. The value is represented by the Dino keyword `char`.
- type of integers. The value is represented by the Dino keyword `int`.
- type of floating point numbers. The value is represented by the Dino keyword `float`.
- type of vectors. The value is represented by the Dino keyword `vector`.
- type of tables. The value is represented by the Dino keyword `table`.
- type of functions. The value is represented by the Dino keyword `func`.
- type of thread-functions. The value is represented by the Dino keyword `thread`.
- type of classes. The value is represented by the Dino keyword `class`.
- type of block instances. The value is represented by the Dino construction `func ()`.
- type of threads. The value is represented by the Dino construction `thread ()`.
- type of objects. The value is represented by the Dino construction `class ()`.
- type of hide values. The value is represented by the Dino keyword `hide`.
- type of hideblocks. The value is represented by the Dino keyword `hideblock`.
- 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 (vector/table element value, variable value, function, thread-function, or class). When the designator referring to table element stands up in the left hand side of an assignment statement, its key becomes immutable.

```
Expr = Designator
```

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

```
Designator = DesignatorOrCall "["  Expr "]"

DesignatorOrCall = Designator
                 | Call
```

The value of the construction before the brackets must be a vector. Otherwise, the exception `indexop` is generated. The value of expression in the brackets (so called the *index*) is converted to integer. If this is not possible, 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:

```
vect [1]
vect ["1"]
vect [1.0]
```

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

```
Designator = DesignatorOrCall "{"  Expr "}"
```

The value of the construction before the figure brackets must be a table. Otherwise, the exception `keyop` is generated. The value of expression in the figure brackets is called the *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, exception `keyvalue` is generated. Examples:

```
tab {'c'}
tab {10}
tab {"1"}
tab {1.0}
```

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

```
Designator = DesignatorOrCall "."  IDENT
           | IDENT
```

Examples:

```
value
value.f
```

5.3 Calls

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

Execution of the body is finished by reaching the block end or by execution of a return-statement. Finishing of the thread-function results in finishing the corresponding thread. The return-statement in a thread-function 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 call returns the value **nil**.

```
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* of operators. 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 C language. 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 the type needed may be made. If after the possible conversions the operands are still not of necessary types, exception `optype` is generated (when something about exceptions in this case is not mentioned). The following conversions may be made by default:

- *Integer conversion*. If the operand is a character, its code becomes integer. 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 integer and is converted into the corresponding integer. 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 32 bits, exception `syserrors.erange`may be generated. In all remaining cases the results of conversion coincide with the operand.
- *Arithmetic conversion*. Analogous to integer conversion except for that the conversion of float pointing number to integer is not made and if the string represents a floating point number (i.e. contains an exponent or fraction), the result will be the corresponding floating point number instead of integer. 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. Analogously if the result is an integer which can not be represented by a 32-bit integer or the result is a floating point number not represented by IEEE double format, the exception `syserrors.erange` may be generated.
- *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 or a floating point number, the result will be a new string of characters which is a decimal string representation of the number.

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 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 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 impilict integer conversion of the operand. If the operand is zero (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, exception `keyop` is generated.

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

Examples:

```
!(type (i) == int & &  type (a) == table & &  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 32-bit integers.

Operators << >>> >> denote correspondingly logical left bit shift, logical right bit shift, and arithmetic (with sign extension) right bit shift of given number (the first operand) by given number of 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. The operators do not generate exceptions (but the conversions may). The operands are equal if they have the same type and equal values (see 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.

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 or floating point type. So the operands should be characters, integers, floating point numbers, or strings representing 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 == 10.0
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 or floating point numbers. Before operator execution, implicit arithmetic conversion is made on the operands. The binary operators `+ - * / %` denote correspondingly 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 or floating point number.

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

Examples:

```
+"0"
+"10."
+"1e1"
```

```
-i
(value + m - 1) / m * m
index % bound
```

5.4.5 Miscellaneous operators

The Dino conditional expression is analogous to the C language one. Implicit arithmetic conversion is made for the first expression followed by `?`. If the value of the expression is non zero (integer or floating point), 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 concatenation of two vectors into a new vector. Before the concatenation implicit string conversion of the operands is made.

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

The operator `char` is used to conversion of a value into a character. First, implicit integer conversion is applied to the operand. The operand should be an integer after the conversion. Otherwise, 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, exception `syserrors.erange` is generated.

The operator `int` is used to 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, exception `optype` will be generated. If the code is too big to be an integer, exception `syserrors.erange` is generated.

The operator `float` is used to conversion of a value into floating-point number. The first, implicit arithmetic conversion is applied to the operand. The operand should be an integer or a floating-point number after the conversion. Otherwise, exception `optype` will be generated. If the result integer is transformed into the corresponding floating-point number. If the code is too big or too small to be a floating-point number, exception `syserrors.erange` is generated.

The operator `vector` is used for conversion of a value into a vector. First, 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 floating point number, or a string. The second parameter value should be a string after implicit string conversion. The format should not be given for a table. The first operand should be a table or a vector after 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 `table` is used to conversion of a value into table. First, 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.

If the operand of the operator `func` is a block instance of the body of a function, it returns the corresponding function. Otherwise, it returns the value **nil**. The operator never generates exceptions.

If the operand of the operator `thread` is a thread, it returns the corresponding thread-function. Otherwise, it returns the value **nil**. The operator never generates exceptions.

If the operand of the operator `class` is an object, it returns the object's class. Otherwise, it returns the value **nil**. The operator never generates exceptions.

```
Expr = Expr "?"  Expr ":" Expr
     | "#"  Expr
     | Expr "@"  Expr
     | type "(" Expr ")"
     | char "(" Expr ")"
     | int "(" Expr ")"
     | float "(" Expr ")"
     | vector "(" Expr ["," Expr] ")"
     | table "(" Expr ")"
     | func "(" Expr ")"
     | thread "(" Expr ")"
     | class "(" Expr ")"
```

Examples:

```
i < 10 ? i : 10
#{"a", 'b'}
#["a", 'b']
"concat this " @ "and this"
type (type)
type (10)
char (12)
vector  (10)
vector  (10, "%x")
vector ({"1":1, "2":2})
table ([1, 2, 3, 4])
func (context (obj))
thread (curr_thread)
class (c ())
```

6 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 C language 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.

6.1 Empty statement

There is also the 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++)
  ;
```

6.2 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 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;
  }
```

6.3 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 section *Variable Declaration*). You can not change the element value of an immutable value (see section *Types and Values*). In this case 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 variable or element of 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.

A special construction `Designator <=> Designator;` swaps values of the designators.

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

```
                      | "-="
                      | "@="
                      | "<<="
                      | ">>="
                      | ">>>="
                      | "& ="
                      | "^="
                      | "|="
```

Examples:

```
v = [10, 20];
i = 1;
i++;
--i;
i *= 20;
v [0] <=> v [1];
```

6.4 Call-statement

A call-statement is used to call a function, a thread-function, or a class. It works analogous to the call in an expression (see sub-section *Calls* in section *Types and Values*). The single difference is in that a call-statement throws away the call's result.

```
ExecutiveStmt = Designator ActualParameters  ";"
```

Examples:

```
putln ("percent=" @ percent @ "%");
newthread ();
```

6.5 If-statement

The Dino if-statement is analogous to the C language one. First, the expression after `if` is evaluated and arithmetic conversion is done to it. The value should be an 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 C language – `else` 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;
```

6.6 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.
2. The expression (*for-guard*) is evaluated and implicit arithmetic conversion is made. The value should be an integer or a floating point number. If this is not true, 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.

If the second statement is a simple statement, the statement semicolon can be omitted. The for-statement also can be finished by execution of the statement `break` in the body. The body can be finished by execution of 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];
```

6.7 Foreach-statement

This statement is used to execution of the foreach-statement body (the statement) for all keys of table which is value of the expression. The expression value should be a table. If this is not true, 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});
}
```

6.8 Break- and continue-statement

Statements `break` and `continue` are used correspondingly to finish execution of the closest-containing for- or foreach-statement covering the statement and to finish one iteration of the body of the for- or foreach-statement. These statement can be used only inside a for- or foreach-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;
```

6.9 Return-statement

Return-statement is used to finish execution of a function, a thread, or class block. The statement corresponds to the closest-containing function, thread-function, or class covering the statement, so the return-statement can be placed only in a function, a function-thread, 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 the default result value **nil**).

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

Examples:

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

6.10 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 an object of a class declared somewhere in predeclared class `except`. If this is not true, exception `optype` is generated. How exceptions are processed is described in the following section.

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

Examples:

```
ext except {
  ext error {
```

```
    class myexcept (msg) {}
  }
}
throw errors.myexcept ("this is an user defined exception");
```

6.11 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 a class declared inside the predeclared class `except`. 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 a class declared inside the predeclared class `except`. If the expression being processed is a class and the exception is an object of the class or an object of a class declared inside 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 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 (invcalls.eof) {
}
try {
  var v = [];
  v {1} = 0;
} catch (except) {
  put ("catching and propagating exception"); println (class (e));
  throw e;
}
```

6.12 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 thread-function call. The thread in which the statement has been executed waits until the expression value becomes nonzero. The expression value (after implicit arithmetic conversion) should be an 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 sync-statement) is executed without interruption by other process. It is used as critical region for process synchronization. In a critical region execution of wait-statement is prohibited (it results in generation of exception `syncwait`). Also thread calls inside a critical region result in generation exception `syncthreadcall`.

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

Examples:

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

7 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 the 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
func 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 = {};
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 (invcalls.eof) {
    break;
  }
func 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 `mpi`:

```
include "mpi";
```

```
var mpi1, mpi2;
mpi1 = mpis.from_string(50, "1000000000000000000000000000000000000");
mpi2 = mpis.from_string(50, "5000000000000000000000000000000000000");
putln (mpis.to_string (mpis.add (mpi1, mpi2)));
```

8 Predeclared identifiers

Dino has quite a lot of predeclared identifiers. The section *Declarations and Scope Rules* contains them in alphanumeric order. Here they are described according to the declaration category which they belongs to.

8.1 Predeclared variables

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

8.1.1 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 arguments to the program (see implementation).
- `env`. The variable value is immutable table whose elements are strings (immutable vectors of characters) representing values of environment variables whose names are the keys of the table.

8.1.2 Version

As Dino is a live programming language, it and its interpreter are in the process of permanent development. To access the Dino interpreter's version number and consequently the language version, the final variable `version` can be used. The variable value is the Dino version as a floating point number. For example, if the current Dino version is 0.54, the variable value will be 0.54.

8.1.3 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 the variable.

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

8.1.4 Exceptions

When it is necessary to create an exception which is a object of a class declared inside class `except` or when it is necessary to refer to a class inside class `except`, the following variables can be used. Instead of typing `catch (except().signal().sigint)`, you could type `catch (signals.sigint)`.

- `excepts`. The variable value is an object of the class `except`.
- `errors`. The variable value is an object of the class `excepts.error`.
- `signals`. The variable value is an object of the class `errors.signal`.
- `invops`. The variable value is an object of the class `errors.invop`.
- `invindexes`. The variable value is an object of the class `errors.invindex`.
- `invkeys`. The variable value is an object of the class `errors.invkey`.
- `invcalls`. The variable value is an object of the class `errors.invcall`.
- `syserrors`. The variable value is an object of the class `invcalls.syserror`.
- `systemcalls`. The variable value is an object of the class `invcalls.systemcall`.
- `invparsers`. The variable value is an object of the class `invcalls.invparser`.
- `invregexps`. The variable value is an object of the class `invcalls.invregexp`.
- `invexterns`. The variable value is an object of the class `invcalls.invextern`.
- `invaccesses`. The variable value is an object of the class `errors.invaccess`.

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

8.1.5 Files

To output something into standard 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.

8.1.6 Miscellaneous variables

Values of the following variables are used by some predeclared functions:

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

8.2 Predeclared classes

The most of the predeclared classes describe exceptions which may be generated in Dino program.

8.2.1 File

Dino has predeclared final class `file`. Work with files in Dino program are made through objects of the class. All declarations inside of class are private. The objects of the class can be created only by predeclared functions `open` or `popen`. If you create an object of the class by calling the class, exception `callop` will be generated.

8.2.2 Exception classes

All Dino exceptions are represented by objects of the predeclared class `except` or of a class in the class `except`. The class `except` has no parameters, therefore all arguments in calling the class will be ignored. There is one predeclared class `error` inside class `except`. All classes corresponding to user-defined exceptions are suggested to be declared in class `except` not in the class `error` because all other exceptions (e.g. generated by the Dino interpreter itself or by predeclared functions) are objects of the class `error` or predeclared classes inside the class `error`. The class `error` and all classes inside the class has one parameter `msg` which contains a readable message about the exception. The following classes are declared in the class `error`:

- `signal`. Classes inside this class 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 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 describes corruption of the Dino program environment (see predeclared variable `env`).

- `invop`. Classes inside this class describe exceptions when operands of operations have an incorrect type or value.
  - `optype`. This class describes that the 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. Now this exception is not generated.
- `invindex`. Classes inside 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.
- `invkey`. 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`. Classes inside this class describe exceptions in calling functions (mainly predeclared ones).
  - `callop`. This class describes that we try to call something which is not a function, class, or thread-function. The exception is also generated when you 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 required type.
  - `invfmt`. This class describes that a format of a format output function is wrong (see function `putf`.
  - `invresult`. This class describes that the result value of function call is not of required type, e.g. comparison function used in a call to function `sort` returns a non integer value.
  - `invinput`. This class describes that the file input is not of required format. Usually the exception is generated by function `scan` etc.
  - `eof`. This class describes that end of file is encountered. Usually the exception is generated by functions reading files (`get`, `scan` etc).
  - `parnumber`. This class describes that the number of actual parameters is not valid when we call a predeclared function.
  - `syserror`. Classes inside this class 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`. Classes inside this class 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`.
- `invparser`. Classes inside this class describe exceptions specific for calling functions of the predeclared class `parser` implementing the Earley parser.

    * `invgrammar`. This class describes the 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 the exception that the parser got an input token with unknown (undeclared) code.
    * `pmemory`. This class describes the exception that there is not enough memory for internal parser data.
  - `invregexp`. Classes inside this class describe exceptions specific for calling predeclared functions implementing regular expression pattern matching.
    * `ebrack`. This class describes the exception that a regular expression has an unmatched bracket.
    * `erange`. This class describes the exception that there is an invalid use of range in regular expression.
    * `ectype`. This class describes the exception that there is an unknown character class name in regular expression.
    * `eparen`. This class describes the exception that a regular expression has an unmatched parenthesis.
    * `esubreg`. This class describes the exception that there is an invalid back reference to a subexpression in a regular expression.
    * `eend`. This class describes the exception that there is a non specific error in regular expression.
    * `eescape`. This class describes the exception that there is a trailing backslash.
    * `badpat`. This class describes the exception that there is invalid use of pattern operators in a regular expression.
    * `esize`. This class describes exception that the compiled regular expression is too big.
    * `espace`. This class describes the exception that there is no memory for a regular expression function to work.
  - `invextern`. Classes inside this class 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 find.
    * `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 externals when they are not implemented under this OS.
  - `invenvar`. This class describes corruption in the type of variables `split_regex` and `time_format` (e.g. their values are not strings).
  - `internal`. This class describes all other (nonspecified) exceptions in calling predeclared functions.

8.2.3 Earley parser classes

Dino has the three following classes which are used by the Earley parser embedded into the Dino interpreter.

Parser. Dino has predeclared final class `parser` which implements the Earley parser. The Earley parser is a very powerful tool to implement serious 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 is to achieve speed and memory requirements which are necessary to use it in prototype compilers and language processors. It parses 30,000 lines of C per second on 500 MHz Pentium III and allocates about 5Mb memory for a 10,000 line C program.
- It makes 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 syntax error recovery. Moreover its error recovery algorithms find error recovery with a minimal number of ignored tokens. It permits implemention of parsers with very good error recovery and reporting.
- It has fast startup. There is practically no delay between processing of grammar and 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.

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 input the parser found that the grammar is ambiguous. The parser can find this even if you asked for only one parser tree (see function `set_one_parse`).
- `set_grammar (descr, strict_p)`. This function tunes the parser to given grammar. The grammar is given by string `descr`. Nonzero value of parameter `strict_p` (after implicit integer conversion) means more strict checking the grammar. 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 exceptions `partype` (if the parameters have wrong types) or `invgrammar` if the description is a bad grammar. The function can also generate exception `pmemory` if there is no memory for 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 name 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.

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 number in the alternative start is 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 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 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 program **dot** of the graphic visualization package **graphviz**. The parameter should be an integer after implicit integer conversion. The function will generate 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 implicit integer conversion. The function will generate exception `partype` if it is not true.

- `set_lookahead (flag)`. This function sets up a flag of of usage of look ahead in the parser work. The usage of lookahead gives the best results with the point of view of 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 function `set_debug`). The parameter should be an integer after 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 default value of the flag is 0. The parameter should be an integer after implicit integer conversion. The function will generate 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` is 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 implicit integer conversion. The function will generate 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 error recovery. The default value is 3. The function returns the previously set up value. The parameter should be an integer after implicit integer conversion. The function will generate exception `partype` if it is not true.

- `parse (tokens, error_func)`. This function is the major function of the class. It makes 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`. If the parser recognizes a syntax error it calls the function given through parameter `error_func` with six parameters:

  - index of the token (in array `tokens`) on which the syntax error occured.
  - the error token itself. It may be **nil** for end of file.
  - index of the first token (in array `tokens`) ignored due to error recovery.
  - the first ignored token itself. It may be **nil** for end of file.
  - index of the first token (in array `tokens`) which is not ignored after 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 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 syntax error was occurred and error recovery was switched off. The function can generate exception `partype` if the parameter types are wrong or exception `invtoken_decl` if any of the input tokens have a wrong code. The function also can generate exception `pmemory` if there is no memory for internal parser data.

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

Token. Dino has a predeclared class `token`. Objects of this class should be the input of the Earley parser (see function `parse` in 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. source line number, name of an identifier, or value for a number).

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

- Terminal node which has reserved name `$term`. The value of the public variable `transl` for this node is an object of class `token` representing the corresponding input token which was an element of the array passed as a parameter of function `parse` of function `parser`.
- Error node which has reserved name `$error`. This node exists in one exemplar (see description of variable `error_anode`) and represents the translation of reserved grammar symbol `error`. The value is public variable `transl` will be **nil** in this case.
- The empty node which has the reserved name `$nil`. This node also exists in one exemplar (see description of variable `nil_anode`) and represents the translation of a grammar symbol for which we did not describe a translation. For example, in a grammar rule an abstract node refers for the translation of a nonterminal for which we do not produce a translation. The value is public variable of such class object will be **nil** in this case.
- Alternative node which has the reserved name `$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 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.

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.

Example of Earley parser usage. Let us write a program which transforms an expression into 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)"`.

```
// 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 parser and set up grammar.
var p = parser ();
p.set_grammar (grammar, 1);

// Add attribute repr to token:
ext token { var repr; }
// The following code forms input tokens from string:
var str = "a+b*(c+d*a)";
var i, inp = [#str : nil];
for (i = 0; i < #str; i++) {
  inp [i] = token (str[i] + 0);
  inp [i].repr = str[i];
}
// The following function output messages about syntax errors
// syntax error recovery:
func 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 translation in polish inverse form
func 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 ();
```

8.3 Predeclared 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, exception `parnumber` is generated. The predeclared functions believe that the actual parameters (may be after implicit conversions) are of the required type. If this is not true, exception `partype` is generated. To show how many parameters the function requires, we will write the names of the parameters and use 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 something is not said about the returned result, the function returns the default value **nil**.

8.3.1 Mathematical functions

The following functions make implicit arithmetic conversion of the parameters. After the conversions the parameters are expected to be of 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 `syserrors.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 `syserrors.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.

8.3.2 Pattern matching functions

Dino has the predeclared functions which are used for *pattern matching*. The pattern is described by *regular expressions* (*regex*). The pattern has syntax of *extended* POSIX (1003.2) regular expressions, i.e. the pattern has the following syntax:

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

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

```
Branch = {Piece}
```

A branch matches a match for the first *piece*, followed by a match for the second piece, etc. If the pieces are omitted, the branch matches the null string.

```
Piece = Atom ["*" | "+" | "?" | Bound]

Bound = "{" Min ["," [Max]] "}" | "{" "," Max "}"

Min = <unsigned integer between 0 and 255 inclusive>

Max = <unsigned integer between 0 and 255 inclusive>
```

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

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

```
Atom = "(" Regex ")"
     | "(" ")"
     | "."
     | "^"
     | "$"
     | BracketedList
     | "\^"
     | "\["
     | "\$"
     | "\("
     | "\)"
     | "\*"
     | "\+"
     | "\?"
     | "\{"
     | "\."
     | <any pair the first character is \ and the second is any
        except for ^.[$()|*+? >
     | <any character except for ^.[$()|*+? >
```

A regular expression enclosed in `()` can be an atom. In this case it matches a match for the regular expression in the parentheses). The atom `()` matches the null string. The atom `.` matches any single character. Atoms `^` and `$` match correspondingly the null string at the beginning of a line and the null string at the end of a line.

An atom which is \ followed by one of the characters `^.[$()|*+?{\` matches that character taken as an ordinary character. Atom which is \ followed by any other character matches the second character taken as an ordinary character, as if the \ had not been present. So you should use \\ for matching with a single \. An atom which is any other single character matches that character. It is illegal to end a regular expression with \. There is an exception which is not described by the atom syntax. An { followed by a character other than a digit or comma is an ordinary character, not the beginning of a bound and matches the character {.

```
BracketedList = "[" List "]"

List = FirstChar ["-" Char] {Char ["-" Char]}

FirstChar = <any character except for ^ - and ]>
          | CollatingElement

Char = FirstChar
     | "^"

CollatingElement = "[:" Class ":]"

Class = "alnum"
      | "alpha"
      | "blank"
      | "ctrl"
      | "digit"
      | "graph"
      | "lower"
      | "print"
      | "punct"
      | "space"
      | "upper"
      | "xdigit"
```

An atom can be a bracket expression which is a *list* of characters enclosed in `[]`. Usually it is used to match any single character from the list. If the list begins with `^`, it matches any single character (but see below) not in the list. If two characters in the list are separated by `-`, this is shorthand for the full *range* of characters between those two (inclusive) in the collating sequence of ASCII codes, e.g. `[0-9]` matches any decimal digit. It is illegal for two ranges to share an *endpoint*, e.g. `a-c-e`.

There are exceptions which are not described by the atom syntax which is used to include a literal `]` in the list by making it the first character (following a possible `^`). To include a literal `-`, make it the first or the last character, or the second endpoint of a range. As you can see from the syntax, all special characters (except for `[`) described in an atom lose their special significance within a bracket expression.

A collating element is a name of a character class enclosed in `[:` and `:]`. It denotes the list of all characters belonging to that class. Standard character class names are:

```
alnum    digit    punct
alpha    graph    space
blank    lower    upper
cntrl    print    xdigit
```

These names stand for the character classes defined in the ANSI C include file `ctype.h`. There is an exception not described by the syntax: a character class can not be used as an endpoint of a range.

There is an extension of regular expressions used by DINO and of ones defined in Posix 1003.2: no particular limit is imposed on the length of the regular expression.

There are the following Dino pattern matching functions:

- `match (regex, string)`. The function searches for the matching regular expression `regex` in `string`. The both parameters should be strings after their implicit string conversion. 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 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 atoms which are regular expressions in parentheses. 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 atom in the regular expression (the atoms are ordered by their open parentheses) and the index of the last character plus one, and so on. If there is no matching with a regular expression in parentheses, the corresponding vector elements will have negative values. Example: The program

  ```
  println (match ("()(a)((a)(a))", "baaab"));
  ```

  outputs

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

- `gmatch (regex, string[, flag])`. The function searches for different occurrences of 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 implicit integer conversion. If its value is nonzero, the substrings matched by regexp 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 index of the first character of substring corresponding to all regular expression in the corresponding occurrences and the second element is index of the last character plus one. If there is no one occurrence, the function returns **nil**. Example: The program

  ```
  println (gmatch ("aa", "aaaaa"));
  println (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 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 atom of the regex in parentheses. 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 by strings matching the regular expression. All parameters should be strings after 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 elements which are the separated substrings. If the regular expression is the null string, the function returns a new mutable vector with elements which are strings each containing one character of string. Examples: The program

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

  outputs

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

  The program

  ```
  println (split ("abcdef", ""));
  ```

  outputs

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

If the regular expression is incorrect, the functions generate one of the following predeclared exceptions (see predeclared classes):

- `ebrack`. Regular expression has unmatched bracket.
- `invregexps.erange`. Invalid use of range in regular expression.
- `ectype`. Unknown character class name in regular expression.
- `eparen`. Regular expression has unmatched parenthesis.
- `esubreg`. Invalid back reference to a subexpression in regular expression.
- `eend`. Non specific error in regular expression.
- `eescape`. Invalid escape sequence in regular expression.
- `ebadpat`. Invalid use of pattern operators in regular expression.
- `esize`. Compiled regular expression is too big.
- `espace`. No memory for the regular expression function to work.

8.3.3 File functions

Dino has some predeclared functions to work on files and directories.

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 implicit string conversion of the parameter). 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').
- `fun (fileinstance_or_filename)`. The function returns new string representing name of 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.
- `fgn (fileinstance_or_filename)`. Analogous to the previous function except for it returns a new string representing 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 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 a parameter is an open file connected to a terminal and 0 otherwise.

The following functions can be used to change rights of usage of the file (directory) for different users. The function expects two strings (after 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 a character 'r', this is right to read (see characters used to denote different rights in the description of the function `fumode`). The functions always return the value **nil**.

- `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.

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

- `readdir (dirpath)`. The function makes implicit string conversion of the parameter value which should be a string (directory path). The function returns a new mutable vector with elements which are strings representing 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 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 parameter value after 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 implicit string conversion) the current directory.

Functions for work with files. The following functions (besides 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 parameter values which should be strings after 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 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 implicit string conversions. The first parameter value is a string representing the file path. The second parameter value is string representing the mode for work with the file (for all possible modes see the ANSI C function `fopen` documentation). All work with an 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 the shell command given by the first parameter value (which should be a string after 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 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 implicit arithmetic conversion and `whence` which should be a string after 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).

File output functions The following functions are used to output something into opened files. All the functions always return the value **nil**. 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 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 strings after 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, exception `parnumber` will be generated. The types of the parameter should correspond to the corresponding conversion specifier (or to be integer for parameterized widths and precisions). If it is not true, exception `partype` will be generated. The format is subset of one of standard C function `printf` and has the following syntax:

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

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

width : '*' | <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, exception `invfmt` is generated.

Flag '#' means that the value should be converted into an alternative form. It can be present only for conversion specifiers 'o', 'x', 'X', 'e', 'E', 'f', 'g', and 'G'. If the flag is used for 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 conversions 'e', 'E', 'f', 'g', and 'G' the output will always contain a decimal point. For 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 flag '#' in format

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

will output

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

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 conversions 'd', 'o', 'x', and 'X' if a precision is given. The flag is prohibited for conversions 'c' and 's'. The following code using flag '0' in format

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

will output

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

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

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

will output

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

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

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

will output

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

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

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

will output

```
->+8 +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 flag '0') on the left (if flag '-' is used) or right. The output is never truncated. The width should be no more than maximal integer value, otherwise exception `invfmt` is generated. The width can be given as a parameter of integer type if '*' is used. If the value of width given by the parameter is negative, flag '-' is believed to be given and the width is believed to be equal to zero. The following code using width in format

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

will output

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

The precision is prohibited for conversions 'c'. If the number after the period is absent, its value will be zero. The precision can be given as a parameter of 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 conversions 'd', 'o', 'x', and 'X' the precision means minimum number of output digits. For conversions 'e', 'E', and 'f' it means the number of 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 precision in format

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

will output

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

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

Conversions 'o', 'x', and 'X' should be used to output an integer value as unsigned in 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 0, the output is empty.

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.

Conversions 'e' and 'E' should be used to output floating point values with an exponent in 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. 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`.

Conversions 'g' and 'G' should be used to output floating point values in 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. 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.

Conversion 'c' should be used to output a character value.

Conversion 's' should be used to output strings.

Conversion '%' should be used to output `%`.

The following code using different conversions in format

```
putf ("->%% %c %s %d %o %x %X<-\n",
      'c', "string", 7, 8, 20, 20);
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<-
->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 string into an opened file associated with a class `file` instance which is the first parameter value.
- `print (...)`. The function outputs all parameter values into standard output stream. The function never makes 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 `func f {}.g(unique_number)` would mean function `f` in the call of function (or class) `g` with the given unique number and function g is in the instance of the implicit block covering the whole program. For the function `g`, output would look simply like `func 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 thread-function `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 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.

File input functions The following functions are used to input something from opened files. All the functions always return the value **nil**. 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 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 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 on 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 on the end of file. The function has an optional parameter which should be integer after 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 character.

- `fget (fileinstance)`. The function is analogous to 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 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.

8.3.4 Time functions

The following functions 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` (integer representing time in seconds since the fixed time) according to the `format` (string). If the format is not given, the value of variable `time_format` is used. In this case if the value of `time_format` is corrupted (it is not a string), the function generates 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.
- %%% - a literal '%' character.

8.3.5 Functions for access to process information

There are Dino 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 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 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.

8.3.6 Miscellaneous 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 or floating point type after implicit arithmetic conversion. So the function can return an integer or floating point number depending on the type of the maximal value after the conversion.
- `min (v1, v2, ...)`. The function is analogous to the previous function, but searches for and returns the minimal value.
- `tolower (str)`. The function expects that the parameter `str` (after implicit string conversion) is a string. The function returns 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 implicit string conversion) is a string. The function returns the new string `str` in which lower case letters are changed to the corresponding upper case letters.
- `trans (str, what, subst)`. The function transliterates charactes in a string. The function expects that the parameters `str` (after 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 <\tt>subst<\tt>. 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.
- `context (par)`. The function returns the context (see 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 thread, a block instance, or an object.
- `inside (par1, par2[, flag])`. The goal for function usage is to check that something is declared inside something other. If the third parameter value after implicit integer conversion is given and nonzero, it is checked with taking contexts into account. The second parameter value should be a function or a class. The first parameter value should be a function, a class, an object, or a block instance. In the first three cases, they define corresponding a function, class, or block. If the function, class, or block defined by the first parameter is declared inside the function or class given by the second parameter, the function `inside` returns 1. The function `inside` also returns 1 if the function or class defined by the first parameter is the same as the function or class 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);
inside (c ().subc (), c ().subc, 1);
```

The first call of `inside` returns 1, while the second one returns 0.

- `subv (vect, index[, length])`. The function is used to extract a slice of vector elements. The first parameter value should be a vector after implicit string conversion. The second and third parameter values should be integers after implicit integer conversion. The function extracts only an element or the part of the slice existing in the vector (so you can use any values of the index and the length). If index is negative, it is considered to be equal to zero. If the length is negative, the slice will finish on the vector end. The function returns a new vector which is the slice. The result vector is immutable only when the original vector is immutable.
- `del (vect, index[, length]) or del (tab, key)`. The first form of the function is used to remove the vector element or a slice of vector elements from the mutable vector `vect`. The second and the third parameter values should be integers after implicit integer conversion. The function removes only an element or the part of the slice existing in the vector (so you can use any values of the index and the length). If index is negative, it is considered to be equal to zero. If the length is negative, the slice will finish on the vector end. The second form of the function is used to remove the element (if it exists) with the given key from the 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.
- `ins (vect, el[, index])`. The function inserts the element given by the second parameter into the vector given by the first parameter on the place given by the third parameter. If the third parameter is not given it is believed to be zero. The third parameter should be an integer after implicit integer conversion. If the third parameter is negative or equal to or greater than the vector length, the element is inserted at the end of the vector. 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])`. 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 reversion of the given vector.
- `cmpv (vect, vect)`. The function makes 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 type). As usual, if this is not true, 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 is in fact generalized lexicographical order.
- `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, or floating point type. If the second parameter is not given, standard arithmetic order (see comparison operators) is used. To use 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.

- `exit (code)`. The function finishes the work of the interpreter with the given code which should be an integer value after implicit integer conversion.
- `gc ()`. The function forces garbage collection and heap compaction. Usually the Dino interpreter itself invokes garbage collection when there is no more free memory.
- `system (command)`. The function executes the command given by a string (the parameter value) in the OS command interpreter. Besides standard exceptions `parnumber` and `partype` the function may generate exceptions `noshell` and `systemfail`.
- `srand ([seed])`. The function sets the parameter value (after implicit integer conversion) as the 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 integer value. If the function `srand` was not called before, 1 will be used as the seed value.
- `sput (...), sputln (...), sputf (format, ...)` The functions are analogous to 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 functions print and println but they return the result string instead of output of the formed string into the standard output stream.

9 Appendix A. Syntax of Dino

```
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
     | Designator
     | INTEGER
     | FLOATINGPOINTNUMBER
     | CHARACTER
     | nil
     | "(" Expr ")"
     | Call
     | "["  ElistPartsList "]"
     | "{"  ElistPartsList "}"
     | STRING
     | char
     | int
     | float
     | hide
     | hideblock
     | vector
     | table
     | func
     | thread
     | class
     | func "(" ")"
     | char "(" Expr ")"
     | int "(" Expr ")"
     | float "(" Expr ")"
     | vector "(" Expr ["," Expr] ")"
     | table "(" Expr ")"
     | thread "(" ")"
     | class "(" ")"
     | type
     | type "(" Expr ")"
     | func "(" Expr ")"
     | thread "(" Expr ")"
     | class "(" Expr ")"

Designator = DesignatorOrCall "["  Expr "]"
           | DesignatorOrCall "{"  Expr "}"
           | DesignatorOrCall ActualParameters
           | DesignatorOrCall "."  IDENT
           | IDENT
```

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

DesignatorOrCall = Designator
                 | Call

Call = Designator ActualParameters

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

VarParList = VarPar { "," VarPar }

VarPar = [final] IDENT [ "="  Expr]

Stmt = ExecutiveStmt
     | Declaration

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

ExecutiveStmt = ";"
              | Designator Assign  Expr ";"
              | Designator ("++" | "--")  ";"
              | ("++" | "--")  Designator ";"
              | Designator ActualParameters  ";"
              | if  "(" Expr ")" Stmt [ else Stmt ]
              | for  "("  Stmt ForGuardExpr ";"  Stmt ")" Stmt
              | for  "("  Designator in Expr ")" Stmt
              | break ";"
              | continue ";"
              | return  [ Expr ] ";"
              | throw  Expr ";"
              | wait  "(" Expr ")" Stmt
              | BlockStmt
              | TryBlockStmt

ForGuardExpr = [Expr]

BlockStmt = Block

TryBlockStmt = try Block { Catch }
```

```
Catch = catch  "(" ExceptClassList ")" Block

ExceptClassList = Expr { "," Expr }

Declaration = VarDeclarations
            | AccessClause
            | ExternDeclarations
            | FuncClassExtDeclaration
            | IncludeDeclaration

VarDeclarations = var  VarParList ";"

ExternDeclarations = extern  ExternItem { "," ExternItem } ";"

FuncClassExtDeclaration = Header Block

AccessClause = (public | private | friend) AccessList ";"

AccessList = IDENT { "," IDENT }

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

ExternItem = IDENT
           | IDENT  "(" ")"

Header = [final] FuncThreadClass IDENT FormalParameters
       | ext IDENT

FuncThreadClass = func
                | thread
                | class

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

Block = "{"  StmtList "}"

StmtList = { Stmt }

Program = StmtList
```

10 Appendix B. Implementation

```
DINO(1)                          User Manuals                          DINO(1)



NAME
```

```
       dino - the interpreter of the programming language DINO

SYNOPSIS
       dino  [  -s  -h  size -Idirname -Lpath -p] (-c program | program-file )
       dino-program-arguments

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

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

OPTIONS
       The options which the DINO interpreter recognizes are:

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

       -h number
              Determine  the  size  of the heap chunks used by the DINO inter-
              preter.  The size can be given in bytes (e.g. 32000),  in  kilo-
              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  col-
              lection.

       -s     Output some statistics of interpreter work into stderr.  Statis-
              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.

       -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 serach for external libraries (if shared or  dll
              libraries  are  implemented  on  the  system.   This is true for
              Linux, Solaris, Irix, OSF, and Windows) in which the Dino exter-
              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.

       -p     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.
       mpi.dll
              the DINO dll library used for implementing MPI on  Windows  sys-
              tems.
       ieee.d
              the  DINO  file implementing IEEE standard floating point arith-
              metic.
       ieee.so
              the DINO shared library used for implementing IEEE on some  Unix
              systems.
       ieee.dll
              the  DINO dll library used for implementing IEEE on Windows sys-
              tems.
       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.
       ipcerr.dll
              the DINO dll library used for  implementing  IPCERR  on  Windows
              systems.
       socket.d
              the DINO file implementing work with sockets.
       socket.so
              the  DINO  shared  library  used for implementing SOCKET on some
              Unix systems.
       socket.dll
              the DINO dll library used for  implementing  SOCKET  on  Windows
              systems.
       There are no temporary files used by DINO.

ENVIRONMENT
       There  are the following environment variables which affect DINO behav-
       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.

DIAGNOSTICS
       DINO diagnostics are self-explanatory.

AUTHOR
       Vladimir N. Makarov, vmakarov@users.sourceforge.net

BUGS
       Please report bugs to cocom-bugs@lists.sourceforge.net.



DINO                             5 May 2001                          DINO(1)
```