Yo: Language Reference Manual

Mengqing Wang, Munan Cheng, Tiezheng Li, Yufei Ou

{mw3061,mc4081,tl2693,yo2265}@columbia.edu

Contents

1 Introduction

Yo is a user-friendly programming language for movie production. We offer the fastest and most efficient non-linear video editing. Users can produce videos from varieties of sources such as images or existing video clips and apply system- or user-defined functions to perform seamless video editing such as clip construction, duration adjustment, subtitle burning. In this light, *Yo*'s objective is to facilitate editing on videos and less human effort needs to be involved.

2 Syntax Notations

In this document, we define types or identifiers in regular expression. The following notations are used to show lexical and syntactic rules.

- Brackets `[ ]` enclose optional items.
- Parenthesis `( )` enclose alternate item choices, which are separated from each other by vertical bars `|`.
- Asterisks $\star$ indicate items to be repeated zero or more times.
- Dash − is a shorthand for writing continuous elements.
- Double colon with an equal sign `::=` is used for definition.
- Braces {n} matches when the preceding character, or character range, occurs n times exactly.
- `{n,m}` matches when the preceding character occurs at least n times but not more than m times, for example, `ba{2,3}b` will find `baab` and `baaab` but not `bab` or `baaaab`. Values are enclosed in braces.

Below we will write *Yo*'s formal syntax definition in gray-background box. The terminals are marked in bold while non-terminals are in regular font.

```
c ::= a | b
```

We will also give examples in the white box.

```
This is an example
```

3 Lexical Conventions

This chapter presents the lexical conventions of *Yo*. This section describes which tokens are valid, including the naming convention of identifiers, reserved keywords, operators, separators and whitespaces.

3.1 Comments

Single line comment is made with a leading `#` in the line:

```
# This is a single line comment
```

Multi-line comment starts with `#{` and ended with `}#`

```
#{ This is a multiline
comment #}
```

Nested comments are not allowed in *Yo*.

3.2 Identifiers

An identifier of *Yo* is a case-sensitive string different from any Keyword (see next subsection). It starts with a letter or an underscore, optionally followed by a series of characters (letter, underscore, number). The length varies from 1 to 256.

Formally, an identifier can be any non-Keyword expressed in regular expression as

```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_]{0,255}
```

Legal examples:

```
_number _number1 number2 number_3 Number
```

Illegal examples:

```
2num num $2 Int Double Bool
```

Note that `Int, Double, Bool` are illegal because they are keywords. A list of keyword can be found in next subsection.

3.3 Keywords

This is a list of reserved keywords in *Yo*. Since they are used by the language, these keywords are not available for naming variable or functions.

break	func	log
continue	global	return
else	if	struct
eval	in	while
for		

Table 1: List of keywords in *Yo*

3.4 Operators

An operator is a special token that performs an operation, such as addition or subtraction, on either one, two, or three operands. A full coverage of operators can be found in a later chapter, See chapter **6 Expression and Operators**.

3.5 Separators

A separator separates tokens. White space (see next section) is a separator, but it is not a token. The other separators are all single-character tokens themselves:
`( ) [ ] ,`

3.6 New Line

A physical line ends with an explicit `\n` input from the user while a logical line contains a complete statement. A logical line can be consist of multiple physical lines, all except the last one ending with an explicit `\`.

```
line 1 \
  line 1 continued \
 line 1 last line
```

3.7 Whitespace

Whitespace characters such as tab and space are generally used to separate tokens. But *Yo* is not a free-format language, which means in some cases, the position and number of whitespaces matters to the code interpretation. Leading tab whitespace is used to denote code blocks and to compute the code hierarchy (similar to curly brackets in C-family languages). Briefly, an extra leading tab lowers the level of this line in the code hierarchy.

In contrast to *Python*, *Yo* only accepts `\t` for leading indent, and space is not allowed. In other words, space should not appear at the beginning of any line (except for a continuing physical line where all the leading whitespaces are ignored).

```
im_a_parent
    im_a_child
        im_a_grandchild
    im_another_child
        im_a_grandchild
```

Usually, `for`,`while`,`if`,`else if`,`else` and function definition may start a new code block. The code block ends with an un-indent. In the above example `im_a_child` and `im_another_child` are at the same code indention level.

4 Types

Yo is a statically and strongly typed programming language, which means the type for each variable, expression or function is determined at compile time and remain unchanged throughout the program.

Yo has an object-oriented model in which every value is an object and each operation is a method call. We have a pure and uniform object model in the sense that the traditional primitive values (integers, double-precision floating numbers) and functions are incorporated into the object model.

The concept of type in *Yo* resembles the *class* in other languages such as C++, Java and Python, which serves as the blueprint for objects. There are three kinds of types: value types, function types and the *None* type. For the sake of definition, we will mention function in this section, but the details will be covered in later sections.

In this section, we first list some built-in types as an introduction to our type system. Then we give the formal definition of type and show how users define types in their program.

4.1 Built-in Types

Below we list the built-in types in *Yo*. As they are used as the building blocks for the program, *Yo* provides *literals* to initialize them conveniently in users' source code. The operators on this types are covered in Section 5.

- `Int` 32-bit signed integral number, ranging from $-(2^{31})$ to $2^{31} - 1$. The literal has to be represented in decimal:

  ```
  IntLiteral ::= [+-]?[1-9][0-9]*
  ```

  The leading plus sign is optional when representing a positive integer. But leading zeros is not legal. For example:

  ```
  86 -320 +300
  ```

  A compile error will be generated if the `Int` literal exceeds range defined above.

- `Double` 64-bit double-precision floating number. The literal is represented as follows:

  ```
  DoubleLiteral ::= [+-]?[1-9][0-9]*.[0-9]+
  ```

  Note that the dot and the fractional number is compulsory (otherwise it can be identified as `Int`). Examples:

  ```
  32.45 -12.0
  ```

- `Bool` Binary value of either `True` or `False`.

  ```
  BoolLiteral ::= True | False
  ```

- `String` A contiguous set of characters. The literal has zero or more characters enclosed in double quotes. A character can be a regular character or an escape sequence. Escape sequences are listed in Table 2.

```
StringLiteral  ::= "StringCharacter"

StringCharacter ::= (^" \ ') StringCharacter
                  | EscapeSequence StringCharacter
```

EscapeSequence	Meaning
\b	Backspace
\t	Horizontal tab
\n	New line
\r	Carriage return
\"	Double quote
\'	Single quote
\\	Backslash

Table 2: Escape Characters

Examples:

```
"abc" "9j32 f0kca0" "Hello\nYo!"
```

- `None` One value type that represents "not exists".

```
NoneLiteral ::= None
```

- `Array` Container holding a sequence of (zero or more) elements. The `Array` literal is represented using a pair of square brackets. Each elements is followed by a comma, and the trailing comma can be omitted.

```
ArrayLiteral ::= [ ArrayElementList ]
ArrayElementList ::= empty
                   | (Literal | Identifier) , ArrayElementList
```

where

```
Literal ::= IntLiteral
          | DoubleLiteral
          | BoolLiteral
          | StringLiteral
          | NoneLiteral
          | ArrayLiteral
```

From the above definition, we can note that the array can be nested. All `Arrays` are considered of the same type (regardless the type of elements it holds). To enumerate a couple of legal examples:

```
[1, 2, 3, "a"] [] ["ab", "ae"] [3.4, 43.0] [None, None]
[[3,2,4], ["ab", "ae"], None]
```

Yo allows users to create an empty `Array` having uniform-typed elements. Adding elements of different types can thus cause a compile error. This can be done by specifying the type name before the pair of square brackets. As an example,

```
Double []
```

returns a empty `Array` exclusively for `Doubles`.

4.2 Value Type Definition

A value type can be defined at the most top level of the program or be nested in another type. The definition starts with the keyword `type` and the type identifier followed by a colon. Conventionally, we use capitalized identifiers for type names.

```
type_def ::= type type_name : NEWLINE INDENT type_element_list DEDENT
type_name ::= identifier
```

Then follows the declaration of zero or more members of value types or function types.

```
type_element_list ::=  empty
          | (value_element_decl | function_element_decl) NEWLINE
              type_element_list
```

A value-typed member is declared by writing the member identifier and its type name or type definition.

```
value_element_decl ::= identifier : type_name
```

For example,

```
zindex: Double
clips: Clip []
```

The type for the member has to be defined in its referencing scope, or a compile error occurs.

```
type A:
    var1: Double
    var2: Int
    var3: A

type B:
    type C:
        var1: Int[]
    var1: A
    var2: Int
    var3: C
```

In the example above, `A` has a member `var3` of its own type `A`. In type `B`, member `var1` is of `A` type (defined on the same level as B) and the type of `var3` is defined inside `B` (`C` is called an *inner type* of `B`). To reference an inner type, users have to use the member operator (see Section 5.8), e.g. `B.C`

The function member defines the executable code that can be invoked about an instance of this type. Details about function definition is covered in Section 7.

4.3 Type Constructor and Object Instantiation

The work of creating an object (or instance) of the type is done in the `eval` function, which is required for every value type. Although the details about function will be elaborated in Section 7, we emphasize here that the `eval` function for a value type has to return an object of this type.

```
type ClipColorMode
    mode: String
    degree: Int
    eval (colorMode: String, colorDegree: Int)
        mode = colorMode
        degree = colorDegree
        return
```

We can instantiate an object of a type in an expression of type name followed by a list of parameters in parenthesis. Formally,

```
type_name ( ParamList )
Paramlist       ::=  parameter (: parameter)* [,]
parameter       ::=  (identifier , type ) | ( paramlist )
```

Here is an example of object instantiation:

```
ccMode = ClipColorMode("RGB", 4)
```

4.4 Frame and Clip

There are two other built-in types: `Frame` and `Clip`. The concepts of *frame*, *clip* and *layer* are as illustrated in Figure 1. A *Clip* is a section of video production. *Clip* is a sequence A *Frame* is seen as one of a sequence still images which compose a *Clip*. It can be constructed directly from an image stored on the hard disk or an extract from an existing *Clip*. Once a *Clip* is assembled from a series of frames at a certain frame rate (usually 24 frames per second), it can be exported as the final product, or to be layered with other *Clip*s to form a new *Clip*.

The constructor of the `Frame` type takes the file name (absolute location) of a picture on the disk. The program fails when there is an error opening the file (e.g. file does not exists).

```
frame = Frame("~/images/pic001.png")
```

The structure of Clip is outlined as below

```
type Clip
    allClips: Clip[]
    startTime: Int          # start time relative to the parent Clip
    length : Double         # Clip original length in seconds
    playTime: Double        # actual play time in seconds
    src: String             # file path if applicable
    func getFrame(frameIndex: Int)  # returns the (frameIndex)^th Frame
    func eval(videoName: String)    # construct from a movie file
    func eval(frameList: Frame[])   # construct from an array of Frames
```

The Clips can be constructed from an existing video on the disk or from a list of Frames. Users can extract a Frame from a clip by specifying the index of the frame. The play speed can be altered by setting the `playTime` and the speedup will be calculated from `length/playTime`.

```
# set 2.5x play speed
clip.playTime = clip.length / 2.5
```

Common operations on `Frame` and `Clip` and be found in section 5.7.

5 Expressions and Operators

This section describes the expression in *Yo*.

5.1 Expressions

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as literals, variables, and function calls that return values.

```
ExpressionStmt ::= Expression
Expression ::= ConditionalExpression
        | ArithmeticExpression
        | FunctionalExpression
```

For details of the definition of function calls, see Section 7.3.

```
42                  # constant
3 * a + 5           # calculation on variables
gcd(20,25)          # a function call
```

Subexpressions can be grouped by parentheses.

```
(1 + 2) * (3 - 4)    # parentheses grouped expressions
```

5.2 Arithmetic operators

Yo provides operators for standard arithmetic operations: addition, subtraction, multiplication, and division, along with modular division and negation. Here are some examples: The two operand must be of same type.

```
x = 4 + 2              # addition          x = 6
y = x - 5              # substraction      y = 1
z = x * y              # multiplication    z = 6
x = 4 + 2.2            # error! 4 is Int but 2.2 is Double
d = -x                 # negation, unary   d = -6
```

For division, if both operands are integers, the result will give the integer part of the quotient. If both operands are doubles, it will return respective mathematical result.

```
w = 7 / 2              # division of integers   w = 3
v = 7.0 / 2.0          # division of doubles    v = 3.5
```

Module operator gives the remainder of a division of two values.

```
d = 10 % 3             # modular divison   d = 1
```

5.3 Array Access operators

The `Array` is indexed and can be accessed using integral subscript starting from zero.

```
a[0]
```

5.4 Comparison operators

Comparison operators are used to determine how the value of two operands relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. The comparison result is either true or false, respectively. The result of such an expression is either true or false.

```
a > b   # a is greater than b
a >= b  # a is greater than or equal to b
a == b  # a is equal to b
a < b   # a is less than b
a <= b  # a is less than or equal to b
a != b  # a is not equal to b
```

5.5 Logical operators

Logical operators test the truth value of a pair of operands. Any nonzero expression is considered true, while an expression that evaluates to zero is considered false.

```
a && b  # and
a || b  # or
!a      # not
```

5.6 Assignment operators

Assignment operators store values in variables. This operator associate from right to left. In the example below, `b` is firstly assigned and then `a` is assigned. After the statement is executed, both `a` and `b` are equal to 42.

```
a = b = 42
```

Formally, an assignment is defined as:

```
AssignmentStmt ::=  (TargetList =)+ (ExpressionList | YieldExpression)
TargetList     ::=  Target (, Target)* ,
Target         ::=  Identifier
                  | ( TargetList )
                  | [ TargetList ]
                  | Attributeref
                  | Subscription
AttributeRef ::=  Primary . Identifier
Subscription ::=  Primary [ ExpressionList ]
Primary ::=  Atom | AttributeRef | Subscription | eval
```

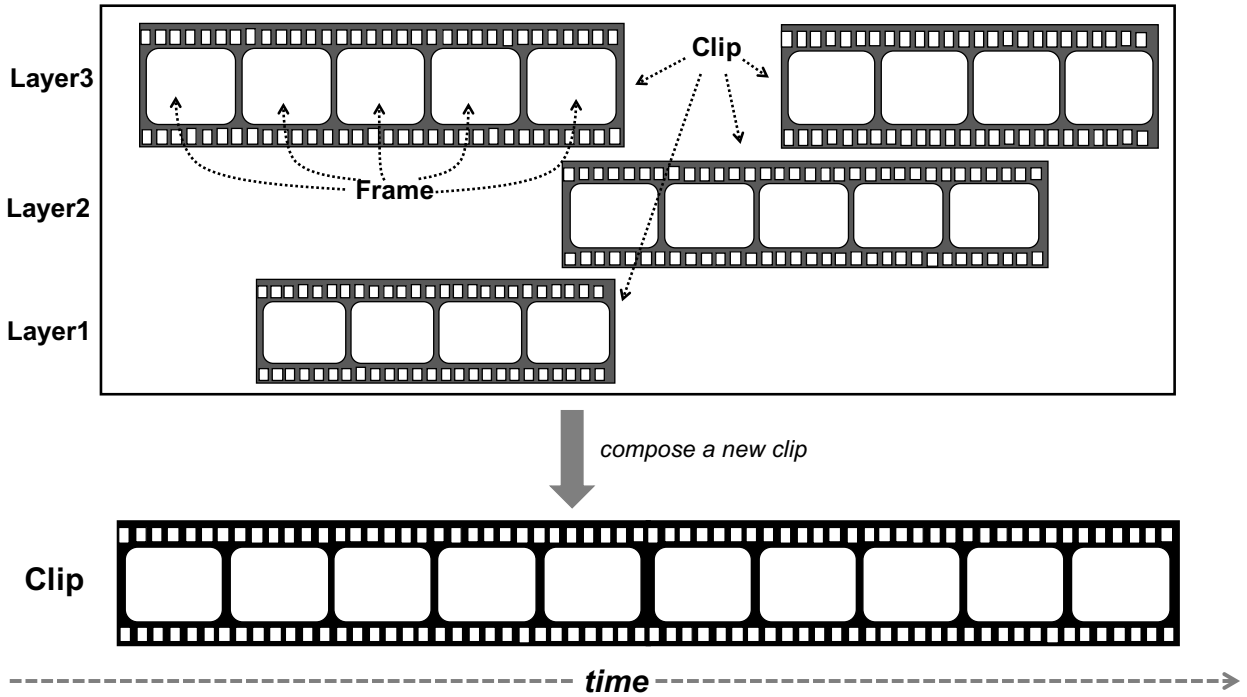

Figure 1: Layering operation

5.7 Clip and Frame operators

Yo allows two clips to be concatenated easily.

```
# concatenate a clip2 to the end of clip1 and form a new Clip
clip3 = clip1 & clip2
```

Yo also subscripts the Clip so that users can extract a range of clip. If the start/end time is not specified, the begin/finish time is used.

```
# get 2.4s - 8.0s of the clip (returns a clip)
clip2 = clip[2.4:8.0]

# get the first 3.2s of a clip
clip2 = clip[:3.2]

# get 2.4s till the end of a clip
clip2 = clip[2.4:]
```

The clip can be layered on the top of another clip to form a new clip. We use a ternary operator, cascade operator, $a\hat{}\ b@c$ to denote putting a on top of b, with time offset of c seconds.

```
# put clip2 on top of clip1, with time offset of 2.4s
clip1 ^ clip2 @ 2.4
```

5.8 Member Access operators

The member access operator `.` is used to access the members of an object: object name followed by the member name

```
# Get a member variable in object myObject
a = myObject.myVariable
# Call a member function defined in object myObject
b = myObject.myFunction(0)
```

5.9 Operator Precedence and Associative Property

When an expression contains multiple operators, such as `a + b * f()`, the operators are grouped based on rules of precedence. For instance, the meaning of that expression is to call the function f with no arguments, multiply the result by b, then add that result to a. The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. Function calls and membership access operator expressions.
2. Unary operators.
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
6. Equal-to and not-equal-to expressions.
7. Logical AND expressions.
8. Logical OR expressions.
9. Clip concatenation.
10. Clip cascading.
11. All assignment expressions.

6 Statements

6.1 overview

A statement could be either a simple statement (using stmt in short) or a compound statement (using compound_stmt in short), formally speaking, they could be defined as:

```
stmt ::= expression_stmt
        | assignment_stmt
        | log_stmt
        | return_stmt
        | continue_stmt
        | break_stmt
compound_stmt ::= if_stmt
        | while_stmt
        | for_stmt
suite ::= stmt NEWLINE | NEWLINE INDENT statement+ DEDENT
```

```
statement ::= stmt NEWLINE | compound_stmt
```

6.2 log

```
log_stmt ::= log expression
```

`log` evaluates the `expression` and writes the resulting object to standard output as a string.

```
log "A string"
# Output: A string

a = 1
log a
# Output: 1

b = 1.01
log b
# Output: 1.01
```

6.3 if

The `if` statement is used for conditional execution:

```
if_stmt ::=  if conditional_expression : suite
             (elif conditional_expression : suite)*
             [else : suite]
```

It selects exactly one of the `suite` by evaluating the `conditional_expression` one by one (start from the conditional_expression next to the `if`, and conditional_expression next to the `elif` sequentially) until one is found to be true; then the `suite` corresponding to this `conditional_expression` is executed, and no other part of the `if` statement is executed or evaluated. If all conditional_expressions are false, the `suite` of the `else` clause, if present, is executed.

```
a = 1
if a:
    log a
    log " is true"
# Output: : 1 is true

a = False
if a:
    log a
    log " is true"
elif !a:
    log a
    log " is not true"
else:
    log "Else clause is executed"

# Output: False is not true
```

6.4 while

The `while` statement is used for repeated execution as long as the `conditional_expression` is true:

```
while_stmt ::=  while conditional_expression : suite
```

This repeatedly tests the `conditional_expression` and, if it is true, executes the `suite`; if the `conditional_expression` is false (which may be the first time it is tested) the loop terminates.

```
a = 0;
while a < 3 :
    a = a + 1;
    log a;
    log " ";
# Output: 1 2 3
```

6.5 for

The `for` statement is used to iterate over the elements of an `array` or continuous integers from `a` to `b`:

```
for_stmt ::=  for target [= a to b | in array] : suite
```

An iterator is created for the result of the expression [= a to b | in array]. The `suite` is then executed once for each item provided by the iterator, in the order of ascending indices. Each item in turn is assigned to the target list using the standard rules for assignments, and then the `suite` is executed. When the items are exhausted (which is immediately when the sequence is empty), the loop terminates.

```
for i = 1 to 3:
    log i
    log " "
# Output: 1 2 3

arr = [1,2,4,6]
for i in arr:
    log i
    log " "
# Output: 1 2 4 6

arr2 = ["1","b","4","c"]
for i in arr2:
    log i
    log " "
# Output: 1 b 4 c
```

6.6 continue

```
continue_stmt ::= continue
```

`continue` may only occur syntactically nested in a `for` loop or `while` loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

```
arr2 = ["1","b","4","c"]
for i in arr2:
    if i=='b': continue
    log i
    log " "
# Output: 1 4 c
```

6.7 break

```
break_stmt ::=  break
```

`break` may only occur syntactically nested in a `for` or `while` loop, but not nested in a function or class definition within that loop. It terminates the nearest enclosing loop.

If a `for` loop is terminated by `break`, since the loop iterator is a local variable, its current value can not be used after the break.

```
arr2 = ["1","b","4","c"]
for i in arr2:
    log i
    log " "
    if i=="b": break
# Output :1 b
```

6.8 return

```
return_stmt ::=  return expression
```

`return` may only occur syntactically nested in a function definition, not within a nested class definition.

`return` leaves the current function call with the `expression` as return value.

```
func foo(a: Int, b: Int):
    return a+b
```

7 Functions

This section discusses the use of functions, which are types in *Yo* and provides the detail for how to declare and define functions, specify parameters and return types and call functions.

7.1 Function definition

A function definition will create a user-defined function object and provide the information below

- the types and values of parameters
- the types and values returned by the function
- the logic composed of a collection of statements that are executed when the function is called

The syntax for a function definition is shown below:

```
function       ::=  func func_name parameter
func_name       ::=  identifier
: NEWLINE INDENT suite DEDENT
paramlist        ::=  parameter (: parameter)* [,]
parameter      ::=  (identifier , type ) | ( paramlist )
```

An example which shows how the function is defined is

```
func FUNCNAME (param1: paramtype1, param2: paramtype2):
    # Function logic goes here
```

In the example, *func* is a keyword which indicates that a function type is defined. Parameters are statically typed with two components: parameter name as an identifier and parameter type as a type. The function type is generated as:

```
type FUNCNAME:
    eval (param1: paramtype1, param2: paramtype2):
        #Function logic goes here
```

The function definition is an executable statement which binds the function name of the local namespaces (defined in Section 8.2) to the function objects. The function objects create a method `eval`. The shorthand for `eval` is to name the instance and follow it with parentheses containing the arguments to the call.

The function definition does not execute the function body until the function object is called, where we define the function call in Section 7.3.

7.2 Parameters and arguments

Parameters are named entities in a function definition that specify arguments that functions can accept. Arguments are values passed to a function object when calling the function, which is defined in the following section. There are two types of arguments defined as follows:

```
positional_arguments ::=  expression (, expression)*
keyword_arguments    ::=  keyword_item (, keyword_item)*
keyword_item         ::=  identifier : type = expression
```

- keyword arguments: an argument preceded by an identifier (e.g. name=) in a function call, for example

```
foo(param1 = 3, param2 = "abc")
```

- positional arguments: an argument which is not keyword argument, for example

```
foo(3, "abc")
```

7.3 Function calls

A function call is defined as:

```
function call        ::=  eval ( argument_list [,] )
argument_list        ::=  positional_arguments [, keyword_arguments]
                          | keyword_arguments [, expression]
```

All argument expressions are evaluated before the call is attempted. If keyword arguments are present, they are first converted to positional arguments. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot. If the slot is already filled, a run-time error will be prompted.

7.4 Function Return Types

A function returns an object as the execution result. The built-in type inference mechanism can allow the omit of certain types. The compiler will deduce the return type based on the body of the function object.

7.5 Recursion

Recursitivity is a property that a function can be called by themselves. The following example shows recursitivity is useful in the calculating an integer's factorial:

```
func factorial (n: Int):
    if n==1
        return 1
    else
        return factorial(n-1)*n
```

7.6 Built-in functions

There are built-in type conversions defined in the following Table 3.

Functions	Conversion rule	Example
Int()	String, Double to Int	»Int("9001") 9001 »Int(8000.0) 8000
Double()	String, Int to Double	»Double("200") 200.0 »Double(10) 10.0
String()	Int, Double to String	»String(100) "100" »String(90.1) "90.1"
Array()	String to Array	»Array("ocamllex") ["o","c","a","m","l","l","e","x"]

Table 3: Built-in type conversion functions

8 Program Structure and Scope

8.1 Program Structure

Yo program must exist entirely in a single source file, with a ".yo" extension. It consists of a number of function/type declarations or statements.

```
program ::= decls EOF

decls ::= empty
        | func_def decls
        | stmt decls
        | type_def decls
```

The position of function/type declarations in the source code does not matter. This means a function can call another defined later and that the member of a user-defined type can use another type defined on the bottom of the source code.

8.2 Namespaces

A namespace *Yo* is a mapping from names to objects. Namespaces include: the set of built-in types (containing functions such as Int()); the global types including built-in types and user-defined types; and the local types in a function invocation. In a sense the set of attributes of an object also form a namespace.

8.3 Scope

A scope is a textual region of a program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace. There are four hierarchies of namespaces in *Yo*:

- Local: the innermost scope, which is searched first, contains the local names

- Enclosed: the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- Global: the next-to-last scope contains the current program's global names
- Built-in: the outermost scope (searched last) is the namespace containing built-in names

Yo follows the rule of Local → Enclosed → Global → Built-in, where the right arrow denotes the namespace-hierarchy search order.

An example shown here gives how different scopes of variable are accessed:

```
p1 = "global variable"

func a_func(p1: String):
    log p1

a_func("local variable")
#p1: local variable
log p1
#p1: global variable
```