Compilers / Compiler Project / Project 2019-2020 / Og Language Reference Manual

*From Wiki ** 3*

< Compilers | Compiler Project

WARNINGS - Evaluation in Normal Season **[Expand]**

Mandatory Use Material **[Expand]**

Contents

/

Og is an imperative language. This manual intuitively presents the characteristics of the language: data types ; name manipulation ; lexical conventions ; structure / syntax ; specification of functions ; instruction semantics ; semantics of expressions ; and finally, some examples .

Data Types

The language is poorly typed (some implicit conversions are performed). There are 5 types of data (4 of them explicitly declarable), all compatible with the C language , and with 32-bit memory alignment:

- Numeric types: **integers** , in addition to two , occupy 4 bytes; the **real ones** , in floating point , occupy 8 bytes (IEEE 754).
- The **strings** are vectors of characters terminated ASCII null (**0x00** , **\ 0**). Variables and literals of this type can only be used in assignments, impressions, or as arguments / returns of functions.
- The **hands** represent address objects and occupy 4 bytes. They can be the subject of arithmetic operations (displacements) and allow access to the indicated value.

The data types can be manipulated individually, using expressions of that type, or in sets (structured type) corresponding to sequences of expressions (tuples). The use of tuples is limited to some situations (see below).

The types supported by each operator and the operation to be performed are indicated in the definition of the expressions .

Name Manipulation

The names (identifiers) correspond to variables and functions. In the following points, the term entity is used to designate them indiscriminately, making explicit when the description is valid only for a subset.

Namespace and visibility of identifiers

The global namespace is unique, so a name used to designate an entity in a given context cannot be used to designate others (albeit of a different nature).

The identifiers are visible from the declaration to the end of the scope: file (global) or function (local). Reusing identifiers in lower contexts masks statements in higher contexts: local redeclarations can cover up global ones until the end of a function. It is not possible to import or define global symbols in function contexts (see global symbols).

It is not possible to define functions within blocks.

Validity of variables

Global entities (declared outside of any function), exist throughout the execution of the program. Local variables for a function exist only during its execution. The formal arguments are valid as long as the function is active.

Lexical Conventions

For each group of lexical elements (*tokens*), the largest sequence of characters is considered to constitute a valid element.

White characters

Separators are considered and do not represent any lexical elements: **line change** ASCII LF (**0x0A** , **\ n**), **indentation of the** ASCII CR **reel** (**0x0D** , **\ r**), ASCII SP **space** (**0x20**) and **horizontal** ASCII HT (**0x09** , **\ t**).

comments

There are two types of comments, which also act as separating elements:

- **explanatory** - start with **//** and end at the end of the line; and
- **operational** - start with **/** * and end with * **/** and may be nested.

If the start strings are part of a character string, they do not start a comment (see definition of strings).

Key words

The words indicated below are reserved (keywords) and cannot be used as identifiers. These words must be written exactly as stated:

```
auto int real string ptr public require sizeof input nullptr
procedure break continue return if then elif else for do write writeln
```

The identifier **og** , although not reserved, when referring to a function, corresponds to the main function , and must be declared public.

Types

The following lexical elements designate types in declarations (see grammar): **int** (integer), **real** (real), **string** (string of characters), **auto** (type inferred from the initial value), **ptr** (pointers). See grammar .

The special type **auto** is used to indicate that the type of the variable or the return of the function must be inferred from the type of its initial value. When applied to a function, it implies that the type of return must be inferred from the expression that is used in the **return** statement(if there are multiple, all must agree on the type to return). This type can be used with a sequence of identifiers, declaring a tuple. The types of the various fields of the tuple are inferred from the corresponding positions of the initial value: it can be a sequence of expressions or another tuple. In both cases, the number of elements must be the same as that of the declared identifiers (ie, even if a variable is used or the return of a function as the initial value for the declared variables, these expressions must be tuples of the same dimension) . Tuples with a single element are identical to the type they contain.

The **auto** type can be used to define generic pointers (such as **void** * in C / C ++), compatible with all types of pointers. They are also the only type of pointer convertible to an integer (the value of the integer is the value of the memory address). The level of nesting is irrelevant in this case, ie, **ptr <auto>** designates the same type as **ptr <ptr <... ptr <auto> ... >>** .

Expression operators

The lexical elements presented in the definition of expressions are considered operators .

Delimiters and terminators

The following delimiters are lexical elements / terminators: **,** (comma) **;** (semicolon), and **(** e **)** (expression delimiters).

Identifiers (names)

They start with a letter, followed by 0 (zero) or more letters, digits or _ (underline). The length of the name is unlimited and two names are distinct if there is a change from uppercase to lowercase, or vice versa, of at least one character.

Literals

They are notations for constant values of some types of language (not to be confused with constants, ie, identifiers that designate elements whose value cannot be changed during the execution of the program).

Whole

An integer literal is a non-negative number. An integer constant can, however, be negative: negative numbers are constructed by applying the unary arithmetic negation operator (**-**) to a positive literal.

Decimal integer literals are made up of sequences of 1 (one) or more digits from **0** to **9** .

Hexadecimal integer literals always start with the sequence **0x** , followed by one or more digits from **0** to **9** or from **a** to **f** (not case sensitive). The letters **a** through **f** represent the values 10 to 15 respectively. Example: **0x07** .

If it is not possible to represent the entire literal on the machine, due to an overflow, a lexical error should be generated.

Floating point reais

Positive real literals are expressed as in C (only base 10 is supported).

There are no negative literals (negative numbers result from the application of the unary negation operation).

A literal without **.** (decimal point) nor exponential part is of the integer type.

If it is not possible to represent the actual literal on the machine, due to an overflow, a lexical error should be generated.

Examples: **3.14** , **1E3** = 1000 (integer represented in floating point). **12.34e-24** = 12.34 x 10^{-24} (scientific notation).

Strings

Strings are enclosed in quotes (**"**) and can contain any characters except ASCII NULL (**0x00** \ **0**). In strings, comment delimiters have no special meaning. If a literal is written that contains **\ 0** , then the string ends in that position. Example: **"ab \ 0xy"** has the same meaning as **"ab"** .

It is possible to designate characters by special strings (starting with \ ), especially useful when there is no direct graphic representation. The special strings correspond to the ASCII characters LF, CR and HT (\ **n** , \ **r** and \ **t** , respectively), quotation mark (\ **"**), forward slash (\\ ), or any other specified using 1 or 2 hexadecimal digits (eg \ **0a** or only \ **a** if the next character does not represent a hexadecimal digit).

Different lexical elements that represent two or more consecutive strings are represented in the language as a single string that results from concatenation.

Examples:

- **"ab" "cd"** is the same as **"abcd"** .
- **"ab" / * comment with "false string"** * **/ "cd"** is the same as **"abcd"** .

Pointers

The only permissible literal for pointers is indicated by the reserved word **nullptr** , indicating the null pointer.

Grammar

The grammar of the language is summarized below. Fixed type elements were considered to be literal; that the curly brackets group elements: (and); alternative elements are separated by a vertical bar: | ; what optional elements are enclosed in square brackets: [and] ; that elements that repeat zero or more times are between ⟨ and ⟩ . Some elements used in grammar are also elements of the language described if represented in fixed type (eg, parentheses).

file	→	*declaration* ⟨ *declaration* ⟩
declaration	→	*variable* **;** | *occupation* | *procedure*
variable	→	[**public** | **require**] *identifier type* [**=** *expression*]
	→	[**public**] **auto** *identifiers* **=** *expressions*

occupation	→	[**public** | **require**] (*type* | **self**) *identifier* **(** [*variables*] **)** [*block*]
procedure	→	[**public** | **require**] **procedure** *identifier* **(** [*variables*] **)** [*block*]
identifiers	→	*identifier* ⟨ **,** *identifier* ⟩
expressions	→	*expression* ⟨ **,** *expression* ⟩
variables	→	*variable* ⟨ **,** *variable* ⟩
type	→	**int** | **real** | **string** | **ptr** **<** (*type* | **self**) **>**
block	→	**{** ⟨ *declaration* ⟩ ⟨ *instruction* ⟩ **}**
instruction	→	*expression* **;** | **write** *expressions* **;** | **writeln** *expressions* **;**
	→	**break** | **continues** | **return** [*expressions*] **;**
	→	*conditional-statement* | *iteration-instruction* | *block*
conditional-statement	→	**if** *expression* **then** *statement*
	→	**if** *expression* **then** *statement* ⟨ **elif** *expression* **then** *statement* ⟩ [**else** *instruction*]
iteration-instruction	→	**for** [*variables*] **;** [*expressions*] **;** [*expressions*] **of** *instruction*
	→	**for** [*expressions*] **;** [*expressions*] **;** [*expressions*] **of** *instruction*

Types, identifiers, literals and expression definition

Some definitions have been omitted from the grammar: data types , *identifier* (see identifiers), *literal* (see literals); *expression* (see expressions).

Left-values

The *left-values* are memory locations that can be modified (except where prohibited by the data type). The elements of an expression that can be used as *left-values* are individually identified in the semantics of the expressions .

Files

A file is called the main file if it contains the main function (the one that starts the program).

Variable declaration

A variable declaration always indicates a data type and an identifier .

Examples:

- Integer: **int i**
- Real: **real r**
- String: **string s**
- Pointer to integer: **ptr <int> p1** (equivalent to **int** * in C)
- Pointer to real: **ptr <real> p2** (equivalent to **double** * in C)
- Pointer to string: **ptr <string> p3** (equivalent to **char** ** in C)
- Pointer to pointer to integer: **ptr <ptr <int>> p4** (equivalent to **int** ** in C)
- Generic pointer: **ptr <auto> p5** (equivalent to **void** * in C)

Global symbols

By default, symbols are private to a module and cannot be imported by other modules.

The **public** keyword allows you to declare an identifier as public, making it accessible from other modules.

The keyword **require** (optional for functions) allows declaring entities defined in other modules in one module. Entities cannot be initialized in these statements.

Examples:

- Declare private variable to module: **real pi = 22**
- Declaring public variable: **public real pi = 22**
- Use external definition: **require real pi**

Startup

When it exists, it is an expression that follows the sign = ("equal"): integer, real, pointer. Real entities can be initialized by whole expressions (implicit conversion). The initialization expression must be a literal if the variable is global.

The strings are (possibly) non - initialized with a null list of values without tabs.

Examples:

- Integer (literal): **int i = 3**
- Integer (expression): **int i = j + 1**
- Real (literal): **real r = 3.2**
- Real (expression): **real r = i - 2.5 + f (3)**
- String (literal): **string s = "hello"**
- String (literals): **string s = "hello" "mother"**
- Pointer (literal): **ptr <ptr <ptr <real> >> p = nullptr**
- Pointer (expression): **ptr <real> p = q + 1**
- Pointer (generic): **ptr <auto> p = q**
- Tuple (simple): **auto p = 2.1**
- Tuple (function): **auto a, b, c, d = f (1)**
- Tuple (sequence): **auto i, j, k = 1, 2, g + 1**

Functions

A function allows you to group a set of instructions in a body, executed based on a set of parameters (the formal arguments), when it is invoked from an expression.

Declaration

Functions are always designated by constant identifiers preceded by the type of data returned by the function. If the function does not return a value, it is declared as a procedure, using the keyword **procedure** to indicate it.

Functions that receive arguments must indicate them in the header. Functions without arguments define an empty header. It is not possible to apply **public** / **require** export / import qualifiers (see global symbols) to a function's argument declarations. You cannot specify initial values (default values) for function arguments. The **auto** type cannot be used to declare argument types (except for defining generic pointers).

The declaration of a function without a body is used to type an external identifier or to make declarations in advance (used to pre-declare functions that are used before being defined, for example, between two mutually recursive functions). If the declaration has a body, a new function is defined (in this case, the keyword **require** cannot be used).

Invocation

The function can only be invoked through an identifier that refers to a previously declared or defined function.

If there are any arguments, when invoking the function, the identifier is followed by a list of expressions delimited by curly brackets. This list is a sequence, possibly empty, of expressions separated by commas. The number and type of current parameters must be equal to the number and type of formal parameters of the function invoked. The order of the current parameters should be the same as the formal arguments of the function to be invoked.

According to the Cdecl convention, the calling function places the arguments on the stack and is responsible for removing them after the call is returned. Thus, the current parameters must be placed in the stack in the reverse order of their declaration (ie, they are evaluated from right to left before invoking the function and the result passed by copy / value). The return address is placed at the top of the stack by the function call.

When the type to be returned by the function to be called is not primitive (ie, it is a tuple), the caller must reserve a memory zone compatible with the type of return and pass it by pointer to the function called as its first argument (see below).

Body

The body of a function consists of a block that can have declarations (optional) followed by instructions (optional). A disembodied function is a declaration and is considered to be undefined.

It is not possible to apply the keywords **public** or **require** (see global symbols) within the body of a function.

A **return** statement causes an immediate interruption of the function, as well as a return of the values indicated as the statement's argument. The types of these values must agree with the declared type. When the type to be returned is not primitive (ie, it is a tuple), the values are copied to the memory area reserved by the caller and passed to the function as a pointer as its first argument (see above).

It is an error to specify a return value for procedures.

Any sub-block (used, for example, in a conditional or iteration statement) can define variables.

Main function and program execution

A program starts with the invocation of the **og** function (without arguments). The arguments with which the program was called can be obtained through the following functions:

- **int argc ()** (returns the number of arguments);
- **string argv (int n)** (returns the nth argument as a string) (**n> 0**); and
- **string envp (int n)** (returns the nth environment variable as a string) (**n> 0**).

The return value of the main function is returned to the environment that invoked the program. This return value follows the following rules (operating system):

- 0 (zero): execution without errors;
- 1 (one): invalid arguments (in number or value);
- 2 (two): execution error.

Values greater than 128 indicate that the program ended with a sign. In general, for correct operation, programs should return 0 (zero) if the execution was successful and a value other than 0 (zero) in case of error.

The run-time library (RTS) contains information about other available support functions, including calls to the system (see also the RTS manual).

Instructions

Unless otherwise indicated, instructions are executed in sequence.

Blocks

Each block has a local variable declaration zone (optional), followed by an instruction zone (possibly empty). It is not possible to declare or define functions within blocks.

The visibility of the variables is limited to the block in which they were declared. Declared entities can be used directly in sub-blocks or passed as arguments to functions called within the block. If the identifiers used to define the local variables are already being used to define other entities within the scope of the block, the new identifier will refer to a new entity defined in the block until it ends (the previously defined entity still exists, but can be directly referred to by name). This rule is also valid for function arguments (see function body).

Conditional statement

This statement behaves identical to the **if-else statement** in C.

Iteration instruction

This instruction has the same behavior as the instruction **for** in C. In the variable declaration area, only an **auto** declaration can be used , in which case it must be the only one.

Termination instruction

The **break** instruction ends the innermost cycle in which the instruction is located, just like the **break** instruction in C. This instruction can only exist within a cycle, being the last instruction in its block.

Continuation instruction

The **continue** instruction restarts the innermost cycle in which the instruction is located, just like the instruction **continues** in C. This instruction can only exist within a cycle, being the last instruction in its block.

Return instruction

The **return** statement , if any, is the last statement in your block. See behavior in the description of the body of a function .

Expressions as instructions

The expressions used as instructions are evaluated, even if they do not produce side effects. The notation is as indicated in the grammar (expression followed by **;**).

Printing instructions

The keywords **write** and **writeln** can be used to present values in the program's output. The first presents the expression without changing the line; the second presents the expression changing lines. When more than one expression exists, the various expressions are displayed without separation. Numeric values (whole or real) are printed in decimal. The strings are printed in native encoding. Pointers cannot be printed.

Expressions

An expression is an algebraic representation of a quantity: all expressions have a type and return a value.

There are primitive expressions and expressions that result from the evaluation of operators .

The following table shows the relative precedence of the operators: it is the same for operators on the same line, with the following lines being of lower priority than the previous ones. Most operators follow C language semantics (except where explicitly stated). As in C, the logical values are 0 (zero) (false value), and different from zero (true value).

Expression Type	Operators	Associativity	Operands	Semantics
primary	**()** **[]**	non-associative	-	curly brackets , indexing , memory reservation

unary	**+ -?**	non-associative	-	identity and symmetrical , indication of position
multiplicative	*** /%**	from left to right	whole, real	C (% is for integers only)
additive	**+ -**	from left to right	integers, reals, pointers	C: if pointers are involved, they calculate: (i) displacements, ie, one of the operands must be of the pointer type and the other of the integer type; (ii) pointer differences, ie, only when the operator is applied **-** to two pointers of the same type (the result is the number of objects of the type pointed between them). If the memory is not contiguous, the result is undefined.
comparative	**<> <=> =**	from left to right	whole, real	Ç
equality	**==! =**	from left to right	integers, reals, pointers	Ç
logical "no"	**~**	non associative	whole	Ç
"it is logical	**&&**	from left to right	whole	C: the 2nd argument is only evaluated if the 1st is not false.
"or" logical	**||**	from left to right	whole	C: the 2nd argument is only evaluated if the 1st is not true.
assignment	**=**	from right to left	all types (except tuples)	The value of the expression on the right side of the operator is stored in the position indicated by the *left-value* (operator's left operand). Integer values can be assigned to actual *left-values* (automatic conversion). In other cases, both types must agree.

Primitive expressions

The literal expressions and invocation of functions have been defined above.

Identifiers

An identifier is an expression if it has been declared. An identifier can denote a variable.

An identifier is the simplest case of a *left-value* , ie, an entity that can be used on the left side (*left*) of an assignment.

Reading

The operation of reading an integer or real value can be performed by the expression indicated by the reserved word **input** , which returns the read value, according to the expected type (integer or real). If used as an argument by the print operators or in other situations that allow various types (**write** or **writeln**), an integer must be read.

Examples: **a = input** (read to **a**), **f (input)** (read to function argument), **write input** (read and print).

Curved brackets

An expression in curly brackets has the value of the expression without the parentheses and allows changing the priority of the operators. An expression in parentheses cannot be used as a *left-value* (see also the index expression).

Expressions resulting from operator evaluation

Pointer indexing

Indexing pointers returns the value of a memory location indicated by a pointer. It consists of a pointer expression followed by the index in square brackets. The result of pointer indexing is a *left-value* .

Example (access to position 0 of the memory zone indicated by **p**): **p [0]**

Tuple indexing

Tuple indexing returns the value of a tuple position indicated by an order number (beginning at 1). It consists of a tuple followed by an integer literal that indicates the position. The result of indexing tuples is a *left-value* .

Example (access to the second position of tuple **a**): **a @ 2**

Identity and symmetrical

The identity (**+**) and symmetric (**-**) operators apply to integers and reals. They have the same meaning as in C.

Memory reserve

The memory reserve expression returns the pointer that points to the memory zone, in the stack of the current function, containing enough space for the number of objects indicated by its entire argument.

Example (vector reservation with 5 reais, indicated by **p**): **ptr <real> p = [5]**

Position indication expression

The suffix operator **?** applies to *left-values* , returning the corresponding address (with pointer type).

Example (indicates the address of **a**): **a?**

Dimension expression

The **sizeof** operator applies to expressions, returning the corresponding dimension in bytes. Applied to a tuple, it returns the sum of the dimensions of its components.

Examples: **sizeof (a)** (dimension of **a**); **sizeof (1, 2)** (8 bytes).

Examples

The examples are not exhaustive and do not illustrate all aspects of the language. Others can be obtained from the course page.

Multi-module program

Definition of the *factorial* function in a file (**factorial.og**):

```
public int factorial (int n) {
  if n> 1 then return n * factorial (n-1); else return 1;
}
```

Example of using the *factorial* function in another file (**main.og**):

```
// external builtin functions
require int argc ()
require string argv (int n)
require int atoi (string s)

// external user functions
require int factorial (int n)

// the main function
public int og () {
  int f = 1;
  writeln "Test for factorial function";
  if argc () == 2 then f = atoi (argv (1));
  writeln f, "! =", factorial (f);
  return 0;
}
```

How to compile:

```
og --target asm factorial.og
og --target asm main.og
yasm -felf32 factorial.asm
yasm -felf32 main.asm
ld -melf_i386 -o main factorial.o main.o -lrts
```

Other tests

Other test packages are available .

Omissions and Errors

Missing cases and errors will be corrected in future versions of the reference manual.

Categories : **Compiler Project** **Compilers** **Teaching**