**Para-C:** **C-like Coding designed to be simple and fast**

LUNA KLATZER, HTBLA Leonding

Draft: 22.06.2021 – Preview v0.1

Introduction

Para-C (From Greek Origin: Beside C[[1]](#endnote-1)) is a programming language designed to integrate C (and other languages), compile to C and serve as an extension to C with additional functionality, simplifications and helper tools. Including adding more features, like new built-in Macros, simplified or new functions, OOP-structures (Under consideration), more straightforward array, list and malloc-handling, expanded data types and additional project-management features.

The compiler will take the Para-C code and compile it down to simple C with the integrated functionality. That means that programming in Para-C will be similar but simpler and well looking due to the simplifications, new structures, keywords and helper functions. Some new structures will not look so new, and possibly like C#, Python or C++, like datatypes, one-liners, overloads and getters etc. To that using Language Extensions, code from other languages can be integrated directly into the code, making it possible to for example use Python inside Para-C, specify the return and convert the datatypes to simple Para-C types, so you can continue coding in Para-C. Using this, you can for example integrate async functionality from Python directly into the program, which is not supported in C and so also Para-C.

Content

[0. Document Style 5](#_Toc75293690)

[1. Base structure 6](#_Toc75293691)

[1.1 Implementation 6](#_Toc75293692)

[1.1.1 Overview Parac Base Library 6](#_Toc75293693)

[1.1.2 Identifiers and Separation of code 7](#_Toc75293694)

[1.2 Structure and file management 7](#_Toc75293695)

[1.2.1 Language Separation 7](#_Toc75293696)

[1.2.2 File Management in the compiled C code 8](#_Toc75293697)

[1.2.3 Modifying the \_\_parac\_\_.c/h file 8](#_Toc75293698)

[1.2.4 Importing PARA-files 8](#_Toc75293699)

[1.2.5 Importing C-libraries or headers 9](#_Toc75293700)

[1.3 Style Conventions 10](#_Toc75293701)

[1.3.1 Naming Conventions 11](#_Toc75293702)

[1.4 The Parac Core library (PCL) 12](#_Toc75293703)

[1.4.1 PCL Imports inside C 12](#_Toc75293704)

[1.5 Name Mangling 12](#_Toc75293705)

[1.6 Built-In Identifiers 12](#_Toc75293706)

[1.6.1 Magic Values in the Para-C source code 13](#_Toc75293707)

[1.6.2 Magic Values in the C source code 13](#_Toc75293708)

[1.7 Running a Program 13](#_Toc75293709)

[1.7.1 Using the CLI 13](#_Toc75293710)

[1.7.2 Project Structure 13](#_Toc75293711)

[1.7.3 Project Configuration 14](#_Toc75293712)

[1.7.4 Entry-File 15](#_Toc75293713)

[1.7.5 Runtime Entry-Point Function 15](#_Toc75293714)

[1.8 In-Code Exceptions 16](#_Toc75293715)

[1.8.1 Keywords 16](#_Toc75293716)

[1.8.2 Usage Examples 17](#_Toc75293717)

[1.9 Types 17](#_Toc75293718)

[1.9.1 The Any-Type 17](#_Toc75293719)

[1.9.2 Storing type vital information 17](#_Toc75293720)

[2. The Para-C Pre-Processor 17](#_Toc75293721)

[3. Compiler and logical Structure 17](#_Toc75293722)

[3.1 Tokens 17](#_Toc75293723)

[3.1.1 C Keywords 17](#_Toc75293724)

[3.1.2 Para-C Keywords 20](#_Toc75293725)

[3.1.3 Special Symbols 20](#_Toc75293726)

[3.1.4 Pre-Processor Directives 21](#_Toc75293727)

[3.2 Structure 22](#_Toc75293728)

[3.3 Lexer and Parser using Antlr4 22](#_Toc75293729)

[3.3.1 Overview 22](#_Toc75293730)

[3.3.2 Pre-Processor Grammar 22](#_Toc75293731)

[3.4 Semantic Analyser (Single-File) 23](#_Toc75293732)

[3.5 File Linker 23](#_Toc75293733)

[3.6 Semantic Analyser (Entire Program) 23](#_Toc75293734)

[3.7 Code Optimiser 23](#_Toc75293735)

[3.8 Code Generator 23](#_Toc75293736)

[3.9 Compiler Warnings 23](#_Toc75293737)

[3.10 Compiler Exceptions 23](#_Toc75293738)

[3.10.1 Error-Codes 23](#_Toc75293739)

[4. Plugins and Extensions 25](#_Toc75293740)

[5. Para-C Language Reference 25](#_Toc75293741)

[5.1 Importing 25](#_Toc75293742)

[5.2 Entry-File Specifier 25](#_Toc75293743)

[5.3 Pre-Processor Directives 25](#_Toc75293744)

[5.4 Functions 25](#_Toc75293745)

[5.5 Variables and Scopes 25](#_Toc75293746)

[5.6 Datatypes 25](#_Toc75293747)

[5.7 Exceptions 25](#_Toc75293748)

[5.8 IO-Interaction 25](#_Toc75293749)

[5.9 Para-C specific Macros 25](#_Toc75293750)

[6. Usage examples 26](#_Toc75293751)

[6.1 Using C-Code inside Para-C 26](#_Toc75293752)

[6.1.1 Using Standard C-code 26](#_Toc75293753)

[6.1.2 Using C-libraries 26](#_Toc75293754)

[6.1.3 Restrictions of Para-C 26](#_Toc75293755)

[Endnotes 27](#_Toc75293756)

# Document Style

**For Proper definition and distinction these styles will be used in the document:**

Standard text:

Normal Font

Name Reference or special name:

Consolas on white background

Code or Value Reference or Snippet:

Italic Consolas with a grey background

Code Block:

Consolas covering the entire line with a grey background

# Base structure

The structure of Para-C will closely lean to the C-Structure (upwards from C11-Standard) but nonetheless have its independent system apart from it. That means it will include its own:

* Structure for its Parac-modules and C-modules (*See* [*File Structure*](#_File_structure))
* Import-structure (Bases on C for compatibility)
* Name mangling (Relative to entry-point of the program)
* Exception handling
* Independent Variable System (Unique types, any type, special return types and union implementations)
* Built-in macros[[2]](#endnote-2)
* Built-in functions[[3]](#endnote-3) (which will partly replace the C-functions for easier handling)
* OOP-Structure using GObject[[4]](#endnote-4) (Under consideration)

## Implementation

*Para-C intends the GNU C Compiler as the primary compiler for the language, however, there is no limitation to use another one, but unexpected results can likely occur.*

The language serves the main purpose of providing new features for the Base-C language, meaning that any structure used will derive from compiled C-code and library-code that provides that functionality. This language library of Para-C is called the Parac Base Library, which is written in C and for higher-level areas Para-C.

### Overview Parac Base Library

*From this point on the Parac Base Library will be referred to as PBL*

The PBL is categorised into three parts:

* Core Library (PCL) – The code required for the Para-C keywords, functions, identifiers and additional core functionality.
* Built-In Library (PBIL) – Built-in functions that are automatically available in the Para-C code (Imported in the C-code).
* Extension Library (PEL) – Extension Functions and identifiers for specific use cases. (Implementations can be at their core C, but the overhead will always be written in Para-C, meaning it will be compiled as well at runtime, but only if it was imported)

The base modules and any additionally used structure will be imported into the project at the top of the file. These imports will be separated from the C-imports of the user which are not associated with the Parac Built-In Library.

### Identifiers and Separation of code

Any PBL identifiers in the C-code will have a clear prefix “\_\_pbl\_” and the suffix “\_\_” (pbl = parac base library), and are going to be signalised if needed, using comments to separate user and compiler code.

*Note: To not be confused with the reserved identifiers in C, the Para-C identifiers have the suffix of two underscores instead of just the regular prefix of one or two which are reserved by the C-conventions. (See C11 – 7.1.3 Reserved Identifiers[[5]](#endnote-5))*

If the Para-C compiler declares new variables, calls new functions or updates values that are not part of the user code, but required for other functionality, such as new keywords, special function calls etc. the compiler will attempt to add a comment with additional information and separate them clearly from the user code.

## Structure and file management

The structure in Para-C is similar to C and works around the compiler directory, the PBL, the compiler libraries and the project module. Therefore, like in C, headers can and should be used for including code from other files. Nonetheless, due to the different structure of Para-C, a project will be partly different from a regular C/C++ project. Especially regarding configuration, where the standard C-Make is replaced by the own Para-C system (The compiled source-code code will contain a CMakeLists.txt file though, but it will be compiler-generated and base on the parac-config.json file configuration (*See* [*Using the parac-config.json file*](#_Using_the_parac-config.json)))

### Language Separation

Since Para-C is based on C, C code can be very simply included in the .parac files. Here a #pragma pre-processor statement will be used to indicate certain code is a specific language. Para-C uses here the PARAC prefix to identify its pragmas.

To indicate a certain portion should be only native C, which will automatically turn off the compiler-generated types, exceptions and logic, the pragma can be used like this:

#pragma PARAC lang C

Here the language is set to C, meaning everything after it will be compiled as if it was C (Watch out for compatibility while using it!). To set back to Parac, use:

#pragma PARAC lang PARAC

### File Management in the compiled C code

In the compiled code managing headers and dependencies will depend on the compiler-generated \_\_parac\_\_.c/h files. The compiler-generated source file \_\_parac\_\_.c will define important identifiers inside Para-C which are used throughout the compiled code, while \_\_parac\_\_.h will define types and macros.

That means the compiler will fetch all required imports, paths and additional data and insert them into the files. These will then be placed at the highest level of the project path hierarchy, meaning the directory of the defined entry-point (*See* [*Using the parac-config.json file*](#_Using_the_parac-config.json)) will be used in the output directory.

### Modifying the \_\_parac\_\_.c/h file

These files can be modified if the user intends to do so. However, it is discouraged to change them, and instead use the non-os-dependent parac-config.json file for general configuration or use the normal .parac source files. *(See* [*Using the parac-config.json file*](#_Using_the_parac-config.json)*)*.

The compiled header file should only be changed if it is necessary for specific changes that need to be done that are not available in the normal parac-config.json file. This is because the \_\_parac\_\_.h is configured for the specific system and compiler where the compilation was run. Import paths will likely or almost always not work on other systems and the program will fail to compile or unwanted results are going to occur.

### Importing PARA-files

The standard in-code importing system will not be different from C and will base on a standard header file. In this header, all publicly available identifiers can be specified, which are then either written in the header itself or the source file (.para). This header can be either included inside your file or another header, which can then be included as well.

For Libraries in Para-C, the simple #include <library.h> can be used, since for C library imports additional commenting is required.

Example:

* Standard Header: #include “header.ph”
* PBL Library Header: #include <library.ph>

#### Renaming included identifiers

The only difference Para-C is introducing is the way you handle the name mangling and possible duplicate identifier names. That means Para-C will introduce new syntax to handle specific cases and “rename” the identifiers to avoid duplicate naming.

The word rename is here in quotation marks since due to the name mangling there can never be the case that a variable has the same name as another variable from another file. Still, in the user code, the mangling is not yet applied meaning that by importing any header containing a variable declaration with an already existing name, the compiler will be unable to identify which variable is the “correct” one.

In this case, you can reorder your code and use a #define macro to point to your variable, but for the sake of easier readability it will use the following syntax for renaming imported values:

#from “<header-file>” include <variable-name> as <new-name>

This will “rename” the variable or signalise the compiler that the new variable name will point to that mangled name. The mangled variable will still be imported in the C-version of the code, but in the Para-C version, it will no longer interfere with existing variables.

### Importing C-libraries or headers

Since Para-C is based on C and backwards compatibility, C-code can be easily included into Para-C, by just either importing the library/header which should be available in the standard C-library or including the header file specified.

To signalise the compiler the header is in native C, the language pragma (*See* [*Language Separation*](#_Language_Separation)) needs to be added before the code-block:

#pragma PARAC lang C

Afterwards to go back to Para-C use:

#pragma PARAC lang PARA

or

#pragma PARAC lang PARAC

Due to combability reasons, the compiler will also go through the included c-header and source code and possibly change minor details to fit the Para-C code. Still, the compiler will not do any major changes and attempt to do as little as possible to keep the functionality alive as wanted.

*Note: Since Para-C is based on C and not too different C++, C++ code may be included, but due to different implementation and unsupported code of C++, it might not work. Use the pragma* #pragma PARAC lang CPP *or* #pragma PARAC lang C++

Example:

#pragma PARAC lang C

#include <stdio.h>

#include <stdbool.h>

#pragma PARAC lang PARA

## Style Conventions

Since Para-C is written in C, style conventions won’t be different in the compiled code or PBL, except the user-specified ones, still inside Para-C naming conventions are a bit different from C for better differentiation of certain types:

* Line-Length Limit is 79 characters for one line
* 4 Spaces per Indentation
* Functions should be declared with the return type definition at the front and the name following in one line. The arguments can be split if it exceeds the line-length limit. Declaring the return type over the name is not allowed, even if it is commonly used around some C-developers.

Example:

void MyFunction(

int arg1, int arg2, int arg3, ...

); // Tab before the arguments

or

void MyFunction(

int arg1,

int arg2,

int arg3,

...

); // Tab before the arguments

* Arrays/Lists should be stretched evenly over multiple lines if the content exceeds the line-limit.

Example:

// One Liner (if it doesn’t exceed the line limit again)

char[] char\_array = {

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ...

}; // Additional one tab

or

// Matrix-Style Array Distribution

char[] char\_array = {

1, 2, 3,

4, 5, 6,

7, 8, 9,

10, 11 ...

}; // Additional one tab in every row

* Indentation level should be around 4/5 to allow readability (Still there is no limitation other than the compiler limitations)

### Naming Conventions

|  |  |  |
| --- | --- | --- |
| **Type** | **Public** | **Internal\*** |
| Header-file | snake\_case |  |
| Source-file | snake\_case |  |
| Structures (Classes, Structs) | PascalCase | \_PascalCase |
| Exceptions | PascalCase (with Error at the end) |  |
| Functions/Methods | PascalCase | \_PascalCase |
| Variables | snake\_case | \_snake\_case |
| Instances | snake\_case | \_snake\_case |
| Constants | SNAKE\_CASE | \_SNAKE\_CASE |
| Types | snake\_case (with possibly \_t at the end) |  |

*\* Internal in this context means variables inside a file, function or structure that should be seen as “private”. Not everything can be internal such as files and exceptions, so in those cases, the field will be empty.*

## The Parac Core library (PCL)

The Para-C core library is, as already explained in the implementation section ([*Implementation of Para-C inside C*](#_Implementation_of_Para-C)), the base for the Para-C programming language. If Para-C functionality is used inside the code that does not exist inside C, the associated core file/library will be imported and used.

This means the user does not have to import any headers themselves, since the compiler automatically will import all core library headers that are needed.

### PCL Imports inside C

The PCL imports won’t be inserted into every file, but inserted into the project-wide header file \_\_parac\_\_.h. This header file will be imported into every resulting C file, meaning if a package is used it will be available in every other file.

For clarification reasons, the compiler will still log errors for imports for unknown identifiers inside Para-C even if they are imported in another file. That means if a library is imported in one file and another file wants to access it without importing it, it will fail due to the compiler not finding the import in the associated file.

## Name Mangling

## Built-In Identifiers

Inside Para-C special macros and reserved identifiers are used to store program-vital data, serve as functions and store configuration data. The identifiers are categorised into library functions, magic identifiers and magic constants. (Similar to the Python magic methods and double underscore definition)

### Magic Values in the Para-C source code

### Magic Values in the C source code

## Running a Program

### Using the CLI

### Project Structure

*Note: To compile or run a program in Para-C, a project setup or configuration is not required since the compiler will compile either way based on an entry file. Still, for organised libraries or programs, it is recommended to use the provided tools to properly manage it.*

In Para-C the Project structure bases on a module-like structure, where a configuration file, called parac-config.json,is used to declare project settings and set general project info. A Project in this case is a simple folder with a configuration file outside of it with a possible readme, .git folder etc.

Possible Look of such a structure:

src/

main.para

main.ph

other.para

other.ph

parac-config.json

LICENSE

README.md

Here the src folder will contain all source files and data required for the program. Inside the parac-config.json the entry-file was set to ./src/main.para, meaning the compiler will start compiling and pre-processing from there and check all included headers and libraries, which are in this case the other.ph and main.ph headers (The compiler includes these two headers because in the main.para file they were included. If they were not, then the compiler would ignore these as they are not visibly needed in the program).

### Project Configuration

The parac-config.json file is the project configuration implementation, which will use CMake in the background. All specified options will be passed to the compiler, which will change behaviour based on the configuration and create the output.

**Possible Configuration:**

* *name* –Name of the Project/Program (Accessible using \_\_*name*\_\_)
* *description*– Description of the Project/Program (Accessible using *\_\_description\_\_)*
* *version*– Version of the Project/Program (Accessible using \_\_*version*\_\_)
* *author*– Author of the Project/Program (Accessible using \_\_*author*\_\_)
* *license*– Distribution License Type for the Project/Program e.g. MIT or GPL (Accessible using \_\_*license*\_\_)
* *entry-point*– Absolute or Relative path to the entry-point file (Must be at the highest level of the project e.g. inside a source folder or next to the parac-config.json file)
* *include*– A list of all files and directories that should be included in the program e.g. data files, configuration files etc. that are not automatically included with the entry-point.
* *compiler-version*– Wanted version of the compiler that should be used. If the version of the compiler used does not match this an exception will be thrown during compilation.

Syntax:

* + *>=0.1* – Version must be greater or equal to 0.1
  + *>~0.1* – Version must be greater than 0.1
  + *<=0.1* – Version must be lower or equal to 0.1
  + *<~0.1* – Version must be lower than 0.1
  + *==0.1* – Version must be equal to 0.1
* *compiler-options*– List of all options that should be used for the Para-C compiler
* *c-compiler-options*– List of all options that should be used for the compiler (In this case it is intended for the GNU C Compiler)

### Entry-File

#### Compilation Specification

Unlike in C, the compiler in Para-C is not designed to be able to compile files without an entry-point or reference on how the program should be run. This is because the compiler goes out from the entry-point file and from there handles all name mangling and imports. This means that to compile a project or file, the entry-file must be explicitly marked as an entry-point. All files and headers used will be compiled as well, but unused files will be ignored even if they are in the same folder, due to uncertainty about how to handle them.

This entry-point file can be either set in the parac-config.json file or using the *parac compile* command where the prompt will ask for the wanted configuration and entry-point.

### Runtime Entry-Point Function

An entry-point in the program is the function that should be called on runtime, to start the entire program. It is not necessarily needed, for example in library code, where the code is imported into another program.

Para-C will also allow pre-compilation of library code, where the user wants to use them in a C-environment and avoid Para-C mangling and runtime handling.

#### Specifying an entry-point function

In Para-C declaring the entry-point with the C-standard *int main()* is not allowed. This is because the Compiler generates the main() function itself based on the program and requirements. Therefore, it uses instead a new syntax where the entry-point function is declared using *entry status <function-name>*.

Example:

entry status Main()

{

// code

}

Entry, in this case, is a new keyword like *static*, which hints to the compiler that this function is the entry-point function. The keyword *status* is here a new built-in datatype, which should be returned from the function. This data type is a struct, which can contain status-code, additional stdout or an entire exception that should be raised. This data type is not restricted to Main alone but can be used inside other functions as well.

*Note: All functions are automatically wrapped using a status return to integrate exception returns and calling stacks. If the code is hinted to be native C, automatically the return is the actual type specified.*

The reason for this seemingly odd decision is partly based on the Para-C language structure, which automatically builds in exception-catching and additional return functionality. The PBL, therefore, provides the entry-point function, which runs the specified entry-point function with additional wrapping and checking that is hidden from the user code. Because of that defining the entry-point does not require the method name to be explicitly Main, meaning a name like *entry status MyProgramMain()* is also valid. However, it is recommended to use it.

## In-Code Exceptions

Para-C provides Exceptions similar to C++, but with a bit of pythonic syntax-sugar added.

These are implemented using compiler-generated compile-types, which automatically implement the user-specified return types. These compile types contain the actual return type and the Para-C return type struct, which defines the exception return.

If an exception was raised, the struct will contain the exception and the call stack. (How call stacks are going to be implemented is uncertain at the moment, but it is expected to use libunwind or libbacktrace. It might be limited though in earlier versions of Para-C)

### Keywords

*raise* – Keyword used to raise an Exception. The following value represents the exception and must be an instance of the exception struct.

*try* – Keyword used to start a try code-block, where exceptions will be passed to the following except statements if the type is included.

*except* – Keyword used to start a handler code-block, which will be called if the expression inside the parenthesis matches the exception type.

*finally* – Keyword used to start a finally code-block, which will be called after all the previous code is executed (including try and except statements). The only exception is if the exception was not included in the except cases or another exception was raised inside an except code-block

*after* – Keyword used to start an after-block, which will be called if no exception is raised in the try-block.

### Usage Examples

## Types

### The Any-Type

### Storing type vital information

# The Para-C Pre-Processor

# Compiler and logical Structure

## Tokens

### C Keywords

*Note: Contains only basic definitions/explanations. For more info see C Keywords [[6]](#endnote-6)*

*auto* – Storage Class Keyword declaring the usage of the default Storage Class in C (Para-C).

*const* – Keyword used to define a Constant Variable that can only be defined once.

*double* – Built-In C Floating Point Datatype (64-Bit).

*float* – Built-In C Floating Point Datatype (32-Bit).

*int* – Built-In C Numeric Datatype (16-Bit or 32-Bit).

*short* – Built-In C Numeric Datatype (16-Bit).

*struct* – Datatype used to declare a Structure Variable or Structure Datatype (if used with typedef).

*unsigned* – Keyword used with Numeric and Floating-Point Datatypes to declare them as unsigned (they can only hold positive values).

*break* – Keyword used to break out of the current loop (Restricted to loops).

*continue* – Keyword used to continue and jump to the next iteration of the loop (Restricted to loops).

*else* – Keyword used to declare an else branch / code-block, which will be called if **all** previous if or else-if statements have failed.

*for* – Keyword used to start a for loop (must be followed by parenthesis containing an expression that either evaluates to *true* (1) or *false* (0)).

*long* – Built-In C Numeric Datatype (64-Bit or 32-Bit, if the used OS is in 32-Bit).

*signed* – Keyword used with Numeric and Floating-Point Datatypes to declare them as signed (they can hold both positive and negative values, used by default).

*switch* – Keywords used to start a switch statement (must be followed by parenthesis containing an expression).

*void* – Built-In C Datatype signalising “nothing”. Used for no-return function or in the context of a pointer (Declaring a variable using void is forbidden).

*case* – Keyword used inside a switch statement to start a case branch. Must be followed by a constant value.

*default* – Keyword used inside a switch statement to start a default branch. Called when all previous cases were *false*.

*enum* – Keyword used to define an Enum Variable containing integer constants

*goto* – Keyword used to jump to a declared label (Not recommended inside Para-C, since it provides unnecessary assembler-like functionality that can be easily achieved using other simpler systems)

*register* – Storage Class Keyword used to hint that a variable *should* be stored inside a register instead of ram. (Very low capacity, but a far better speed on read and write)

*sizeof* – Built-In Function which can be called to get the size of a specified variable. (Returns the size in bytes)

*typedef* – Keyword used to create a new custom type

*volatile* – Keyword used to signalise that the keyword can be changed in an unspecified way by the hardware. This also means that the compiler will not do any optimisations based on the logic of the program, since the value might change during runtime even if the compiler does not see that.

*char* – Built-In C Datatype representing a char / numeric value for a char (4-Bit)

*do* – Keyword used to start a do-while loop code-block.

*extern* – Storage Class Keyword used to define a global variable that is visible to all object modules. It will point to the address of the prior declared variable with the same name and can not be initialised.

*if* – Keyword used to declare an if-branch / code-block, which will be called if the statement inside the parenthesis evaluates to *true*.

*return* – Keyword used to return from the current function to the caller of the function. Can contain a value if the function-type is not *void*

*static* – Storage Class Keyword used to define a static variable, which will retain its value until the end of the program. This means if a static variable is declared inside a function and the value is increased inside that function, calling the function multiple times will use the old value and contain from there, meaning the old value will **not** bereinitialised. (Similar to constant, but with the difference, the value can be changed)

*union* – Union Keyword used to create a union. The union can contain multiple types and will reuse the same storage for all types, meaning that the biggest type inside the union will define the size of it. If initialised only one value can be used at the same time.

*while* – Keyword used to start a while loop (must be followed by parenthesis containing an expression that either evaluates to *true* (1) or *false* (0)).

### Para-C Keywords

*entry* – Keyword used to hint the compiler at the entry point function of the program. This can be only used once and multiple usages will raise an error during compilation.

*status* – Built-In datatype which represents an exit status for a program.

*raise* – Keyword used to raise an Exception. The following value represents the exception and must be an instance of the exception struct.

*try* – Keyword used to start a try code-block, where exceptions will be passed to the following except statements if the type is included.

*except* – Keyword used to start a handler code-block, which will be called if the expression inside the parenthesis matches the exception type.

*finally* – Keyword used to start a finally code-block, which will be called after all the previous code is executed (including try and except statements). The only exception is if the exception was not included in the except cases or another exception was raised inside an except code-block

*after* – Keyword used to start an after-block, which will be called if no exception is raised in the try-block.

### Special Symbols

|  |  |
| --- | --- |
| Arithmetic Symbols | Addition(+), Subtraction(-), Modulo(%), Multiplication(\*), Division(/) |
| End of line | ; |
| Sequencing | , |
| Code-Block | { } |
| Subexpression Grouping | ( ) |
| Assignment | = |
| Special Assignment | +=, /=, \*=, -=, %=, &=, |=, ^=, <<=, >>= |
| Increasement or Decrement | ++, -- |
| Condition Evaluation | Questionmark(?), Colon(:) |
| Comparison | ==, !=, <, <=, >, >= |
| De- or Reference | \*, &, [ ] |
| Member Selection | ->, Dot(.) |
| Pre-processor Directive | # |
| Bit-Operator | &, ~, |, ^ |
| Decorator Specifier | @ |
| Logical | &&, ||, ! |
| Shift Operator | >>, >>>, <<, <<< |

### Pre-Processor Directives

|  |  |
| --- | --- |
| #define | Substitutes a pre-processor macro. |
| #include | Inserts a particular header from another file. |
| #undef | Undefines a pre-processor macro. |
| #ifdef | Returns true if this macro is defined. |
| #ifndef | Returns true if this macro is not defined. |
| #if | Tests if a compile-time condition is true. |
| #else | The alternative for #if. Following Code is inserted if the previous if was evaluated as *false* |
| #elif | #else and #if in one statement. (adds another possible branch) |
| #endif | Ends pre-processor conditional. |
| #error | Prints error message on stderr. |
| #pragma | Issues special commands to the compiler, using a standardized method. |
| defined(…) | Can be used inside #if or #elif to check whether an item is defined or not defined. Logical Operators are allowed between items inside define() |

## Structure

## Lexer and Parser using Antlr4

### Overview

Para-C uses for easier parsing Antlr4, which automatically generates a Python folder containing the parser, listeners and lexer based on the *ParaC.g4* file defining the grammar of the language. The generated code will be wrapped inside a module, which will call and use the generated code.

Using that, it will convert on runtime the inserted file into a list of logic trees, which contain all needed information for statements. These logic trees are then returned and used to compile the program.

### Pre-Processor Grammar

Since the C Pre-Processor is syntax-wise different than the standard C language, mainly noticeable due to the line endings (pre-processor statements with hard-line endings (\n or \r\n, while the normal lines use declared line breaks in form of a Semicolumns(;)), the Para-C Pre-Processor (See [The Para-C Pre-Processor](#_The_Para-C_Pre-Processor)) is similarly structured. Meaning that the grammar is split between normal statements and preprocessor statements.

Meaning that Pre-Processor statements are defined inside Antlr4 as lexer rules instead of parser rules, meaning that complex wrong syntax usages will not be reported, since it is treated as a whole token. This is specifically true with #define, which can contain any expression after the Macro name specifier. Basic errors will be thrown though in specific cases, like for example *#undef* where logically only an identifier is allowed, meaning the lexer will throw an error if a number or an expression is used.

## Semantic Analyser (Single-File)

## File Linker

## Semantic Analyser (Entire Program)

## Code Optimiser

## Code Generator

## Compiler Warnings

The Compiler while running will check for basic information and will report on possibly problematic issues such as logical issues, possible loss of data or problematic usages of certain types. While running these will be counted and at the end of the run logged as a summary of the process.

To that, syntax warnings for non-fatal formatting and inconsistency issues can be reported, as a help to avoid causing inconsistent writing and style. This also includes the partly stricter conventions, that try to improve on the loose C-conventions, which are more open to writing code.

## Compiler Exceptions

Exceptions inside Para-C are categorised into two categories:

* Non-Fatal Exceptions, which do not interfere with continuing to check the file and
* Fatal Exceptions, which can not be ignored and cause the compiler to interrupt the process and exit.

If only Non-Fatal Exceptions get noticed by the compiler, the compilation will finish with a summary containing a counter for all errors and warnings while running the compilation.

If a Fatal-Issue is received and causes a hard interrupt while running, the return code will be an error code that is specified here. This helps for better categorisation for certain errors. An error message will also appear with the error code at the end of the file, including a trace-back if the issue is a bug inside the compiler.

### Error-Codes

* 1\*\* Internal Errors:

100 – Internal Error: An Exception in the Internal parts of the compiler that are not related to the compilation.

101 – Interrupt Error: The compiler received an interrupt while running. (Derives from the Python Base Exception KeyboardInterrupt)

102 – Config Not Found: The parac-config.json file for the project was not found, which is responsible for configuring the Project and compiler.

103 – Antlr4 Compiled files not found: The antlr4 lexer and parser files were not found

* 2\*\* User Input Exceptions:

200 – User Input Error: General Exception due to faulty input of the user

201 – File Permission Error: Failed to access (read, write) to existing file due to missing permissions

202 – File not Found: The File was not found and does not exist! If the file can’t be seen it will be treated as well as a File not Found Error.

203 – Is Directory: File is a directory

* 3\*\* Lexical Error:

300 – Lexical Error: An issue occurred in the Tokenizer / Lexical Analyser step of compiling. (Derive from the Antlr4 lexer errors)

* 4\*\* Parser Error:

400 – Parser Error: An issue occurred in the Parser (Logic Tree generator), which tries to convert the generated Antlr4 tokens into proper Logical Para-C tokens

* 5\*\* Logical Error:

500 – Logical Error: An issue occurred while walking through the program, which was caused due to logical irregularity and incompatible statements.

* 6\*\* Linker Error:

600 – Linker Error: An issue occurred while linking the files together and checking dependencies and mergeability. (Logical issues like double declarations or importing a name that was already defined will be treated as linker error since they directly result from the linking process)

* 9\*\* Other Errors:

900 – Other Error: Exception of type other that is assignable to any other type of exception

901 – Unknown Error: Received an unknown exception while running.

# Plugins and Extensions

# Para-C Language Reference

*(This part of the document serves as the reference for all keywords, identifiers, functions etc. that are added in Para-C and will provide information on how to properly use them)*

## Importing

## Entry-File Specifier

## Pre-Processor Directives

## Functions

## Variables and Scopes

## Datatypes

## Exceptions

## IO-Interaction

## Para-C specific Macros

# Usage examples

## Using C-Code inside Para-C

### Using Standard C-code

### Using C-libraries

### Restrictions of Para-C

Endnotes

1. Para meaning and origin: [[link]](https://en.wiktionary.org/wiki/%CF%80%CE%B1%CF%81%CE%AC#Preposition) [↑](#endnote-ref-1)
2. List of Pre-defined macros in C: [[link]](https://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html) [↑](#endnote-ref-2)
3. List of Built-in C-functions: [[link]](https://www.tutorialspoint.com/ansi_c/c_function_references.htm) [↑](#endnote-ref-3)
4. Introduction to GObject: [[link]](https://www.freedesktop.org/software/gstreamer-sdk/data/docs/latest/gobject/howto-gobject.html) [↑](#endnote-ref-4)
5. C11– Draft April 12, 2011: [[link]](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf) [↑](#endnote-ref-5)
6. C Keywords: [[link]](https://en.cppreference.com/w/c/keyword) [↑](#endnote-ref-6)