**Para-C:** **C-based Coding for Advanced Management of Programs and Code**

LUNA KLATZER, HTBLA Leonding

Draft: 29.06.2021 – Preview v0.0.1

Introduction

Para-C (From Greek Origin: Beside C[[1]](#endnote-1)) is a programming language that is designed to integrate other languages and allow for advanced management of programs / code-bases inside a program, where the language will serve as a base overhead language with extended C-functionality and simplifications to write simpler code. Including adding more features, like new built-in functions, libraries, data structures, decorators, function-handling, OOP structures (Under consideration), and additional project-management features.

To achieve the multiple language “support” / integration-functionality, the compiler will take the Para-C code and compile the source code down to simple C and generate the code required to integrate the wanted language, using their required compiler/interpreter for the language. That means that programming in Para-C will be similar to C and partly C#, due to the new features, keywords and helper functions, but add the simple option to integrate and manage simple code or even programs that should be directly embedded into the program. Using this, you can for example embed async functionality from Python directly into the program, which is not natively supported, and then pass generated data to a C++ program, which then uses that to run something else.

Content

[0. Document Style 6](#_Toc75623810)

[1. The Para-C Core Language 7](#_Toc75623811)

[1.1 Overview 7](#_Toc75623812)

[1.2 Implementation 7](#_Toc75623813)

[1.2.1 Overview Parac Base Library 7](#_Toc75623814)

[1.2.2 Identifiers and Separation of code 8](#_Toc75623815)

[1.3 Structure and file management 8](#_Toc75623816)

[1.3.1 Language Separation 8](#_Toc75623817)

[1.3.2 File Management in the compiled C code 9](#_Toc75623818)

[1.3.3 Modifying the \_\_parac\_\_.c/h file 9](#_Toc75623819)

[1.3.4 Importing PARA-files 9](#_Toc75623820)

[1.3.5 Importing C-libraries or headers 10](#_Toc75623821)

[1.4 Built-In Identifiers 11](#_Toc75623822)

[1.4.1 Magic Values in the Para-C source code 11](#_Toc75623823)

[1.4.2 Magic Values in the C source code 11](#_Toc75623824)

[1.5 Name Mangling 11](#_Toc75623825)

[1.6 In-Code Exceptions 11](#_Toc75623826)

[1.6.1 Keywords 12](#_Toc75623827)

[1.6.2 Usage Examples 12](#_Toc75623828)

[1.7 Data-Types 12](#_Toc75623829)

[1.7.1 The Any-Type 12](#_Toc75623830)

[1.7.2 Storing type vital information 12](#_Toc75623831)

[1.8 The Parac Core Library 12](#_Toc75623832)

[1.8.1 PCL Imports inside C 13](#_Toc75623833)

[1.9 Style Conventions 13](#_Toc75623834)

[1.9.1 Naming Conventions 14](#_Toc75623835)

[1.10 Running a Program 14](#_Toc75623836)

[1.10.1 Using the CLI 14](#_Toc75623837)

[1.10.2 Project Structure 14](#_Toc75623838)

[1.10.3 Project Configuration 15](#_Toc75623839)

[1.10.4 Entry-File 16](#_Toc75623840)

[1.10.5 Runtime Entry-Point Function 17](#_Toc75623841)

[1.10.6 Specifying an entry-point function 17](#_Toc75623842)

[2. The Para Language Extensions 18](#_Toc75623843)

[2.1 Overview 18](#_Toc75623844)

[2.2 Syntax 18](#_Toc75623845)

[2.3 Process Management 19](#_Toc75623846)

[2.4 Return Data 19](#_Toc75623847)

[3. The Para-C Pre-Processor 19](#_Toc75623848)

[3.1 Overview 19](#_Toc75623849)

[3.2 Usage of Pre-Processor Directives 19](#_Toc75623850)

[4. The Para-C Compiler 19](#_Toc75623851)

[4.1 Overview 19](#_Toc75623852)

[4.2 Structure 20](#_Toc75623853)

[4.3 Lexer and Parser using Antlr4 20](#_Toc75623854)

[4.3.1 Overview 20](#_Toc75623855)

[4.3.2 Basic Structure (Parsing Tree Components) 20](#_Toc75623856)

[4.3.3 Pre-Processor Grammar 20](#_Toc75623857)

[4.4 Semantic Analyser (Single-File) 21](#_Toc75623858)

[4.4.1 Overview 21](#_Toc75623859)

[4.4.2 Algorithmic structure 21](#_Toc75623860)

[4.5 File Linker 21](#_Toc75623861)

[4.6 Semantic Analyser (Entire Program) 21](#_Toc75623862)

[4.7 Code Optimiser 21](#_Toc75623863)

[4.8 Code Generator 21](#_Toc75623864)

[4.9 Compiler Warnings 21](#_Toc75623865)

[4.10 Compiler Exceptions 22](#_Toc75623866)

[4.10.1 Error-Codes 22](#_Toc75623867)

[4.11 Tokens 23](#_Toc75623868)

[4.11.1 C Keywords 23](#_Toc75623869)

[4.11.2 Para-C Keywords 26](#_Toc75623870)

[4.11.3 Special Symbols 26](#_Toc75623871)

[4.11.4 Pre-Processor Directives 27](#_Toc75623872)

[5. Core Language Reference 28](#_Toc75623873)

[5.1 Importing 29](#_Toc75623874)

[5.2 Entry-File Specifier 29](#_Toc75623875)

[5.3 Pre-Processor Directives 29](#_Toc75623876)

[5.4 Functions 29](#_Toc75623877)

[5.5 Variables and Scopes 29](#_Toc75623878)

[5.6 Datatypes 29](#_Toc75623879)

[5.7 Exceptions 29](#_Toc75623880)

[5.8 IO-Interaction 29](#_Toc75623881)

[5.9 Para-C specific Macros 29](#_Toc75623882)

[6. Usage and Implementation Examples 29](#_Toc75623883)

[6.1 Using C-Code inside Para-C 29](#_Toc75623884)

[6.1.1 Using Standard C-code 29](#_Toc75623885)

[6.1.2 Using C-libraries 29](#_Toc75623886)

[6.1.3 Restrictions of Para-C 29](#_Toc75623887)

[Endnotes 30](#_Toc75623888)

# 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

# The Para-C Core Language

## Overview

The Para-C Core Language is the fundamental language and base from where language extensions will derive from. It is an optimised and updated version of the C11-Standard, which implements improved concepts and handling for better management and usability. It is usable using the Para-C Compiler, which will transpile the Para-C code into C-code or directly binary executable. (*See* [*The Para-C Compiler*](#_The_Para-C_Compiler))

## Implementation

*Para-C intends the GCC 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 Para-C 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 functionality Para-C (will be compiled on runtime as well).

### 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) – Mainly, but not necessarily, language extensions, functions for specific use cases and specific identifiers. (Non-Langauge extensions 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. In the case of language extensions, they will have two libraries, one Para-C and one in the target language that is integrated)

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[[2]](#endnote-2))*

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.

## 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.

## The Parac Core Library

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.

## Lexical Conventions

## 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 |
| Property field (struct or simple class) | snake\_case |  |
| Variables | snake\_case | \_snake\_case |
| Instances | snake\_case | \_snake\_case |
| Constants | SNAKE\_CASE | \_SNAKE\_CASE |
| Types | alllowercase (this does not apply to returns or specific types of return) |  |

*\* 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.*

## Declaration and Definitions

## Operators

## Selection Statements

## Iteration Statements

## Namespaces and file management

### 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.

The only difference Para-C is introducing is the way you handle variable names and name mangling. That means Para-C will introduce new syntax to handle specific cases and provide the option if needed to rename identifiers to avoid duplicate naming.

#### Importing entire headers

For Libraries in Para-C, the simple #include <library.h> can be used, but for C library imports additional commenting is required. (*See* [*Importing C-libraries or headers*](#_Importing_C-libraries_or))

Example:

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

This will, unlike in C, *not* include all identifiers from that header, but rather tell the compiler a specific header reference that should be used inside the file. This means the user will need to specify the wanted header or specify imported variables (Explained in the next section).

To specify the header the namespace syntax (*See* [*Namespaces*](#_Namespaces) *for more info*) needs to be used:

headername::variable

#### Specifying imported variables

To specify what variable should be imported or recognised inside a file, a specific syntax can be used:

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

In the case, the variable-name is already taken and the user wants to “rename it”, the following pre-processor directive can be used.

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

This will signalise the compiler that this new variable name will point to that mangled name and every occurrence should be replaced with the mangled name/the actual name in the included header. 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

### Namespaces

### Language Separation

Since Para-C is based on C, C code can be very simply included in the .parac files. For that, the Compiler will just require specific hinting for the language and the compiler will correctly process them. For that, 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

### Dependency and Header Management in the 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.

## Built-In Types

### The Any-Type

### Storing type vital information

## 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

## Casting and Type-Conversion

## Enumerators

## Functions

### Function Overloading

### Decorators

### Ellipsis

## Structures and Simple Classes

## Lambda Functions

## Pointers

## 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

## Name Mangling

Para-C applies, like C++, name mangling to its variables and identifiers, the only exception being pre-processor directives or defines, meaning that the user can generate two functions with the same name, but different parameter types (also called function overloading). To that, names that are included from other files will be wrapped with the specific file name and place to differentiate it from variables with the same name. This allows that imported names are renamed, or referenced using their parent header name (*See for more info* [*Importing PARA-files*](#_Importing_PARA-files))

# The Para Language Extensions

## Overview

The Para Language Extensions are extensions for integrating other programming languages into Para-C. Using Extension tasks, a program can be embedded into the Para-C Program and called with parameters. The Program will wait until the Process has finished and return its returned data if they are given.

These Processes are compiled with the main program, meaning the binaries are created and able to be shipped with the entire Project. The only requirements are that the compiler or interpreter of the language is added and that the compiler knows how to embed the process into the Para-C program. This is done using the installable extensions and libraries inside the language.

## Syntax

The Language Extension Syntax is relatively simple and consists of the extension task declaration that defines code from another language that should be embedded and the call for that task.

### Declaring an extension task

To declare an extension task this syntax is used:

*exttask <lang> <identifier> (<args>) {config} <Optional: lambda>*

The identifier in this case is like a regular function that can be called inside the Para-C program. After that, the arguments, which will be passed onto the called file or code, and the configuration, which defines how the compiler should treat the extension task, are declared.

If inside the configuration no file was specified a lambda function (*See* [*Lambda Functions*](#_Lambda_Functions)) needs to be created which directly adds the code to the program. This lambda block is then turned into a function in the specified language, which will be stored inside a compiler-generated file in the output folder.

*exttask* <lang> <identifier> *(<args>)* {config} <lambda>

Otherwise, if a file should be imported/used, it will need to be specified in the parameters, including the entry-point function. (Depends on implementation and extension. Every extension can / will have its way of managing this, due to the logical differences in languages) If no file is used, they will default to NULL.

*Example (lambda):*

*e*xttask python TestFunc1(int x, int y, int z) {

file: NULL,

file\_entry: NULL,

} => {

// Insert code of the language

};

*Example (included file):*

*e*xttask python TestFunc2(int x, int y, int z) {

    file: "file.py",

    file\_entry: "main"

};

### Calling an extension task

To call a task, the following syntax is used:

spawn <func\_name>(<args>);

To catch the return simply assign the value to a variable:

ExtTaskReturn <variable\_name> = spawn <func\_name>(<args>);

## Process Management

## Return Data

# The Para-C Pre-Processor

## Overview

The Para-C Pre-Processor is, like in the GCC C Compiler, a component besides the core language that will be called to interpret and process pre-processor directives that alter the source code or interact with the compiler. These directives use a Hashtag(#) as the prefix for the directives (*For a list of directives see* [*Pre-Processor Directives*](#_Pre-Processor_Directives)).

## Algorithm and Parsing

The Pre-Processor utilises, like the core language, Antlr4 as the Parser and Lexer, but in this case, the Para-C code will be “ignored” and only the pre-processor directives are going to be parsed and then used to process the file. Meaning the Pre-Processor will use the context and the stored lines to create the resulting file/files.

To that, a major difference to the core language is that while processing; all files are going to be directly included in the process and checked to correctly apply pre-processor effects. (This will be done over the *#include* directive)**.** These files will then also be passed to the Compiler, which will use them in the Linker (*See* [File Linker](#_File_Linker_1)) and Multi-File Semantic Analyser (*See* [*Multi-File Semantic Analyser*](#_Semantic_Analyser_(Entire)) to correctly analyse included files as well.

After finishing the parsing and modification process, the pragmas and the generated altered code will be passed to the Compiler. Meaning that in the Compiler, pre-processor directives will not be visible anymore and error logs will display the altered code, meaning it will be visible how the pre-processor interpreted the specific directives and altered the code.

## Usage of Pre-Processor Directives

# The Para-C Compiler

## Overview

The Para-C Compiler is the Compiler responsible for the Para-C Core Language (*See* [*The Para-C Core Language*](#_The_Para-C_Core)) and the Para-C Language Extensions (*See* [*The Para Language Extensions*](#_The_Para_Language)). It parses, analyses and links the files together to generate a C-source code output or using the GCC C Compiler (GNU Compiler Collection C Compiler) an executable.

*Note: If language extensions are used, the compiler generates executables for each language extension (Even if multiple files and components are used, the compiler will merge them into a single program, which will, depending on the program, call different modules and code), meaning that the output will contain multiple executables. There are tools though that merge these into a single executable, to make shipping the code easier.*

## Structure

The steps of a compilation process (Flowchart)

The Structure of the Compiler is made up of multiple components and modules, which have specified jobs assigned e.g. components will not interact with each other and the compiler will use the components to put all pieces together, which in the end make up the finished program.

The different modules are:

* The Pre-Processor, which will alter the code based on the pre-processor directives (*See* [*The Para-C Pre-Processor*](#_The_Para-C_Pre-Processor_1))
* The Lexer and Parser, which generate the tokens and the logic trees of the program. (Includes the Conversion to Logical Tokens, which simplifies the tokens to make the job easier for the Semantic Analysis. *See* [*Lexer and Parser using Antlr4*](#_Lexer_and_Parser))
* The Single-File Semantic Analyser (*See* [*Single-File Semantic Analyser*](#_Semantic_Analyser_(Single-File)))
* The File Linker (*See* [*File Linker*](#_File_Linker_1))
* The Multi-File (Program-Wide) Semantic Analyser (*See* [*Multi-File Semantic Analyser*](#_Semantic_Analyser_(Entire))
* The Code Optimiser (*See* [*Code Optimiser*](#_Code_Optimiser))
* The Code Generator (*See* [*Code Generator*](#_Code_Generator))

## Lexer and Parser

### Overview

The Para-C Core Language uses for easier parsing Antlr4, which automatically generates python source files 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.

The Lexer and Parser will start by first parsing the main file and then afterwards all included files (Data of the included files will be passed by the Pre-Processor). These will then be wrapped and passed to the Semantic Analyser (Both Single and Multi-File).

### Antlr4 Structure (Parsing Tree Components)

The basic structure of the Parsing Tree is based on the compilationUnit, which describes the entire file and the EOF (End of File Character). The main component, one layer under that, is the translationUnit, which describes the actual code that will be processed in the compiler. The translationUnit contains a list of externalItems, which can be either stray Semicolumns(;), definitions or function definitions.

### 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)) uses its grammar file, lexer and parser, which will correctly parse everything and pass it onto the Compiler. (For more info see [Algorithm and Parsing](#_Algorithm_and_Parsing))

## Single-File Semantic Analyser

### Overview

The Semantic Analyser (Single-File) is the first step after the parsing process, which will introduce logical checking on the file and validate whether statements are valid or not. This is limited though in certain cases, specifically identifiers, which are not defined inside the file, since there can be the case of an identifier being imported and so the Single-File Analyser can not check for the existence in other files. (This step is moved to the [*File Linker*](#_File_Linker) and [*Semantic Analyser (Entire Program)*](#_Semantic_Analyser_(Entire)*).*

### Algorithmic structure

The Semantic Analysis will go through each externalDeclaration (Can be either a declaration or functionDefinition) and go through each block individually and check for logical correctness. If a function is called, both external function definition and Para-C function definition, the Compiler will only validate arguments and correct usage of the function. Pre-Processor statements will be already ignored and ‘removed’ in this state since the Pre-Processor already altered the code appropriately.

## File Linker

## Multi-File Semantic Analyser

## 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.

## Tokens

### C Keywords

*Note: Contains only basic definitions/explanations. For more info see C Keywords [[3]](#endnote-3)*

*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. (Can either be a single *value* or *(expression)*) |
| #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() |

# Core Language Reference

This part of the document serves as the reference for all keywords, identifiers, functions etc. and changes in Para-C. The main focus will be to provide examples and information on functionality, changes and syntax.

## Importing

## Entry-File Specifier

## Pre-Processor Directives

## Functions

## Variables and Scopes

## Datatypes

## Exceptions

## IO-Interaction

## Para-C specific Macros

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. C11– Draft April 12, 2011: [[link]](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf) [↑](#endnote-ref-2)
3. C Keywords: [[link]](https://en.cppreference.com/w/c/keyword) [↑](#endnote-ref-3)