Development Manual

Prerequisites:

- The transpiler creates a parser using grammar and lexer files for C++ using ANTLR. The grammar file `CPP14Parser.g4` and lexer file `CPP14Lexer.g4` are taken from https://github.com/antlr/grammars-v4/tree/master/cpp . You may supply your own grammar and lexer files as well (but it might change the structure of the parser completely). The grammar and lexer files are present in `Grammar/`

- Other files taken from the link are `CPP14ParserBase.py`. This is present in `dist/`

- It requires a python environment for running the transpiler. Preferably anything above python 3.10. It also requires a Java runtime installed for gui purposes which is automatically installed when installing antlr (visualising the parsetree)

- There is only a single dependency (antlr) which can be installed by running:
```
pip install antlr4-tools
pip install antlr4-python3-runtime
```

- The ParserVisitor can be generated by running the following commands:
```
antlr4 -Dlanguage=Python3 Grammar/CPP14Lexer.g4 -visitor -o dist
antlr4 -Dlanguage=Python3 Grammar/CPP14Parser.g4 -visitor -o
dist
```

- Rust Compiler and formatter

Approach, Working and Debugging:

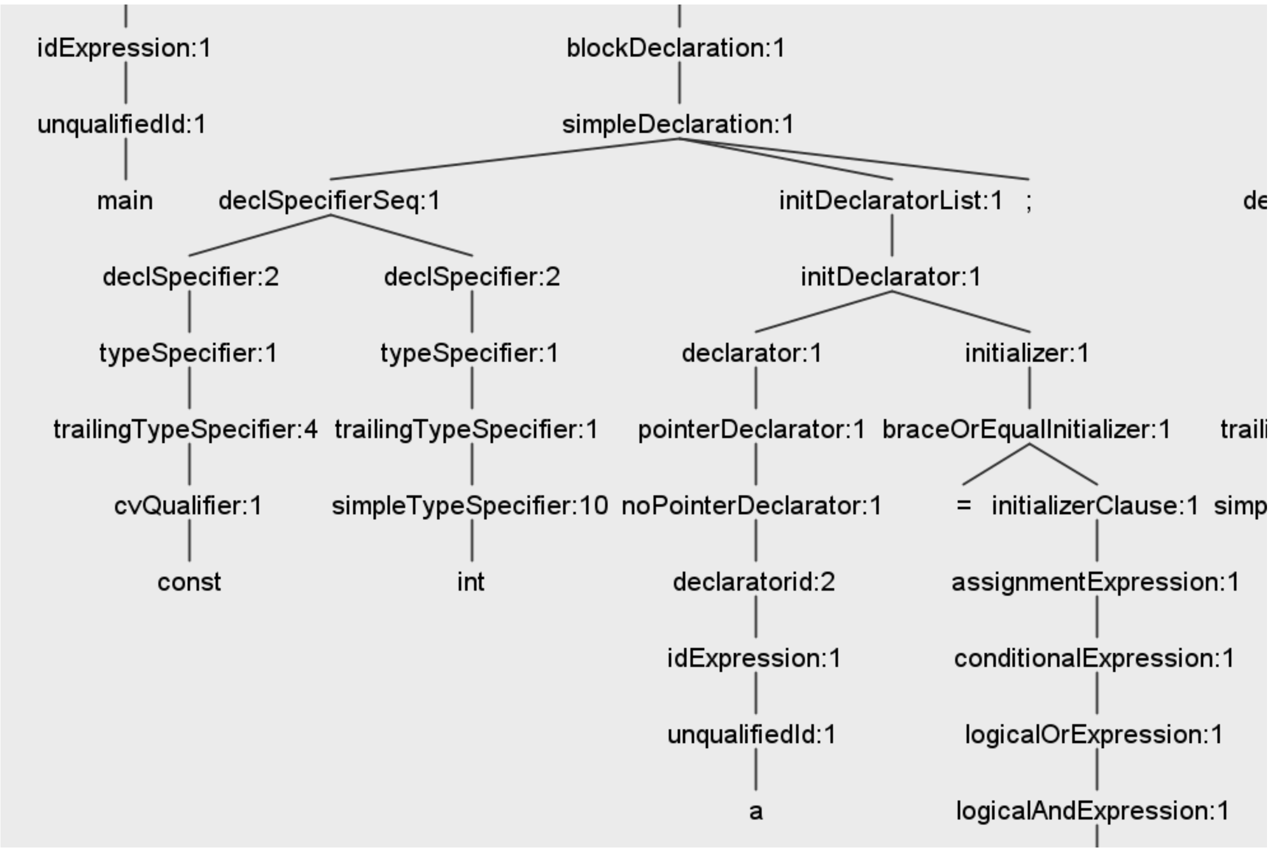
We use a parser-visitor approach to traverse the parsetree. Antlr generates a `dist/CPP14ParserVisitor.py` program from the grammar and lexer files. This has a single class `class CPP14ParserVisitor(ParseTreeVisitor)` . We inherit this base class and override the necessary visitor functions in `CPPtoRustConverter.py.` The flow of method calls is starting from the start symbol of the grammar : `translationUnit` hence `visitTranslationUnit()` method is called , which in turn, calls the visitor functions of the children non terminal nodes by the `self.visitChildren(ctx)` call. This order of invocation follows a depth-first traversal of the parse-tree.

The driver code is `main.py` which imports `CPPtoRustConverter` class and takes two command line arguments . The usage for the same is **python main.py <filename> <debugging-level>** . The <debugging-level> argument is optional and defaults to 0 (false) . It can be explicitly set to 1 for debugging purposes (illustrated later)

The attribute `self.rustCode` is responsible for storing the currently transpiled rust code. Inside the visitor functions we aim to generate equivalent rust code corresponding to the subtree of the corresponding Non-terminal symbol in the parsetree.

It is not a trivial task to deduce which functions we need to modify to support a particular construct. It helps to visualise the parsetree of a particular short program to quickly note which Non-Terminals are responsible for generating the construct in question. The parsetree can be generated by the command `antlr4-parse Grammar/CPP14Parser.g4 Grammar/CPP14Lexer.g4 translationUnit -gui < $filename`

where $filename can be replaced by a cpp file of your choice. A sample parsetree looks like:



Note that not all code generation for a block of code occurs at the visitor function of that particular Non-Terminal only. It may very well be handled further up in the parse tree where the context of a larger block of code may be necessary to effectively generate code for constituent parts (for example Casting , where the type of LHS is necessary).

This might make it harder to track where code generation is happening. We can use the debugging level set to 1 to see which methods are responsible for adding which tokens in the generated code. This can be inferred by inspecting the `log.csv` file which is written by the `logToCsv(current_function_name, ctx_text, rust_code)`. This writes the row of data to a csv file.

This can help us find quickly which method was responsible for adding a particular token during code generation. This can make both modification and fixing bugs easier

sample `log.csv` entries:

	A	B	C

406	expressionVisitor	"%d"	#![allow(warnings, unused)]
			fn main ( ){
			const a :i32 = 5 as i32;
			let mut b = 4 * 2 + 1 ;
			let mut c :char = 'a' as char;
			let mut d :char;
			let mut d :bool = true as bool;
			let mut e :i64 = 1 as i64;
			let mut e :u64 = if d { 5 } else { 6 } as u64;
			print! (

407	visitInclusiveOrExpression	"%d"	#![allow(warnings, unused)]
			fn main ( ){
			const a :i32 = 5 as i32;
			let mut b = 4 * 2 + 1 ;
			let mut c :char = 'a' as char;
			let mut d :char;
			let mut d :bool = true as bool;
			let mut e :i64 = 1 as i64;
			let mut e :u64 = if d { 5 } else { 6 } as u64;
			print! (

Approach for Handling STL Containers:

- Instead of transpiling each method call for commonly used containers by mapping corresponding method calls in cpp to rust. (For example for std::vector::push_back() to std::Vec::push())
- We write a struct in a module in rust (that can be imported from `libs/`) that supports the same methods as the cpp counterparts and uses the rust methods internally. This way the method calls can be transpiled as is (though in some cases some modification might be necessary as in swap or operator=)
- The only part requiring significant change would then be the declaration of the objects.

Sample Transpilation:

Declaration1	`vector<int> v1;`	`let mut v1 = vector::new().clone();`
Declaration2	`vector<int> v2 = {1, 2, 3};`	`let mut v2 = vector![1, 2, 3].clone();`

Testing:

The testing method used is to first transpile C or C++ code into rust using the transpiler and then checking for rustfmt and compiler errors. A Makefile is included in the base directory and can be used to run tests. The tests/ folder contains C or C++ files which act as regression tests. Running ***make test*** will test all the files in the tests/ folder for compilability.