**MINEFUNK COMPILER**

Stages of compilation:

* Parsing
* Pre-index check
* Indexing
* Post-index check
* Re-index I
* Translation I
* Re-index II
* Translation II
* Optimization

Each stage of compilation is performed on every input file. For example, the parser is run of every input file before the pre-index checker is run of every AST tree.

**Parsing**

Parsing is the process of converting the raw input files into abstract syntax trees (ASTs). The input stream is first split into tokens (a set of “words” useful to the lexical analyser). The lexical analyser then converts the stream of tokens to an abstract syntax tree. Syntax errors are reported during this stage of compilation. Minefunk uses the JavaCC parser generator to generate its parser and JJTree to generate the AST classes.

**Pre-index check**

Pre-index checking checks for invalid things in various AST nodes which are too complicated for the parser to check for. For example, it checks if a function has valid modifiers. Any compiler errors found are reported.

**Indexing**

The indexer goes through all AST trees and makes a note of all types, fields and functions defined. The index is global, so all members will be visible from all files. If any duplicate members are found, a compiler error is reported.

**Post-index check**

The post-index checker performs all the checks that the pre-index checker was not able to do without the index. This includes checking that a function call calls a function which actually exists. The post-index checker should also check for circular variable references (e.g. *int a = a;*). Any compiler errors found are reported. After this point, all the compiler errors should have been found and subsequent stages of compilation can assume that all input is valid.

**Re-index I**

At this point all type, variable and function names and references are replaced with IDs. There will be no distinction between local variables and fields after this point. However, information will be retained about where the variable was declared (required to decide if a variable is recursive).

**Translation I**

The first stage of translation translates the high-level AST tree into a lower-level intermediate language. This greatly simplifies subsequent processes because the intermediate language is more relevant to the final Minecraft commands. It will translate loops into recursive function calls, blocks inside if statements into their own functions, and more. Type information will be erased (as will the types themselves).

**Re-index II**

During this stage, the compiler will attempt to reduce the number of variables and functions to the smallest number possible. This is done by re-using variables (even between separate functions) and inlining non-recursive functions which are only called once.

**Translation II**

This is when the intermediate language will finally be translated into lists of raw commands, or structures with one-to-one equivalence.

**Optimization**

Finally, the optimizer attempts to reduce the overall number of Minecraft commands by omitting redundant code. For example:

*testfor @e[tag=foo]*

*scoreboard players tag add @s flag {SuccessCount:1}*

*function bar:baz if @s[tag=flag]*

can be reduced to

*function bar:baz if @e[tag=foo]*