Z Programming Language

Programming Languages

2023/2024

Index

Introduction

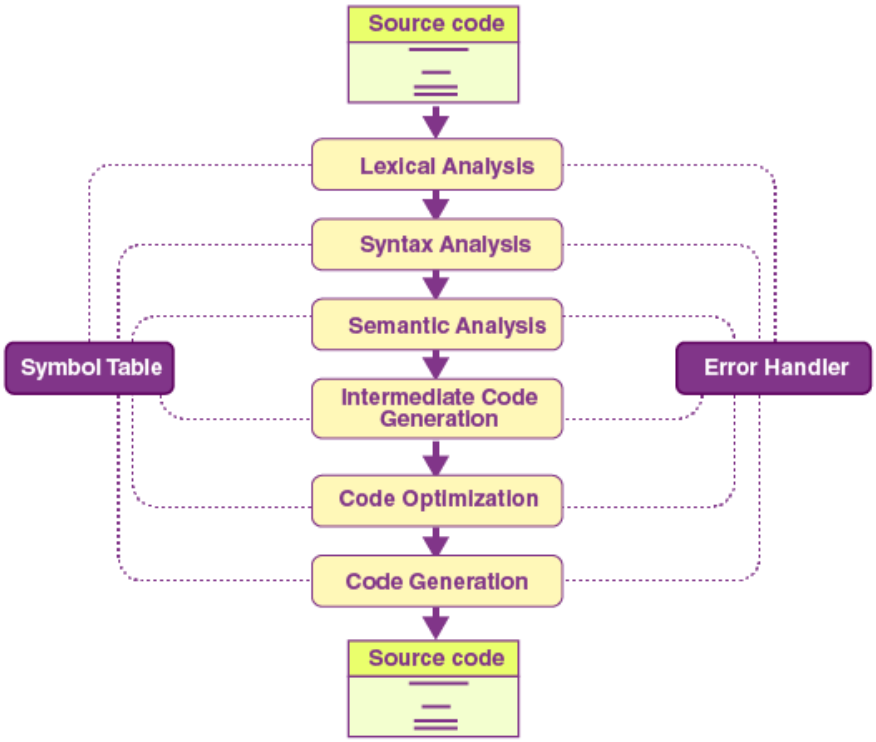
Welcome to Z, a programming language designed for the digital-native generation: Gen Z. Inspired by the unique slang of Gen Z, Z is engineered to be user-friendly and accessible, motivating a generation known for its technological prowess and innovation to delve into the world of programming. Z is more than just a programming language, it's a reflection of who we are, it speaks our language literally! It embraces our technology dependency, turning it into fuel for innovation. In Z, your code doesn't just run, it vibes.

Our attention span is short, but our dreams are big. Z is about making coding quick, fun, and engaging. Breaking away from the traditional syntax of most programming languages, allowing us to learn and use, even for those with no prior coding experience. It’s simple, straightforward syntax encourages creativity and experimentation, allowing users to express their ideas quickly and intuitively.

Let’s build a better future together, one line of code at a time!

Compiler Modules

The process followed by our compiler to transform the high-level language source code into machine code is structured in the following six modules, split between two distinguished sections: front-end and back-end.



Front-end

This first section of the compiler is responsible for analyzing the given source code to validate that it meets the requirements by checking any possible mistakes in the different modules, raising a corresponding error if it doesn't. If the code is confirmed to be correct, then an intermediate representation is generated and sent to the next stage. Below you will find a more detailed explanation of each of the included modules.

Pre-compiler

The pre-compiler is a stage that comes before the compilation process strictly starts and it is used to make different kinds of modifications to the code to prepare it for the compilation. Such as expanding macros and removing comments.

```
<< Comment >>
```

In our case the pre-compiler is in charge of removing comments and removing the tabs from the code, so the source code is transformed into a sequence of strings, blank spaces and end of lines. Making it as small and compact as possible so the stages to come analyze the code as fast as possible.

Lexical Analyzer

The Lexical Analyzer module, also known as scanner, describes the first phase of the actual compiling process. The scanner is tasked with generating a sequence of tokens from the raw source code. This phase serves as the foundation for subsequent stages by categorizing and associating data to each element in the source code.

The Syntax Analyzer retrieves sequentially each token, thus, the scanner offers a *nextToken* function which fetches the next token from the tokenizer. A custom tokenizer is used extending java's *StringTokenizer* to consider, for instance, line-column location in source code and specific literals as a single token.

Every time the next token is requested, the custom tokenizer fetches the next token according to the following delimiters:

```
¡!¿?()%.,[]:&|\"'\n
```

If it contains ‘ or “, it means that we stand before a char or a string literal, respectively. Thus, we will continue fetching the next tokens until we find another ‘ or “ indicating the end of the literal. All tokens gathered are concatenated to treat it as a single token. If we are outside of a literal definition, all spaces and newlines are ignored in a loop until a valid token is found.

We also have defined a special delimiter set to allow treating these tokens as a single one: ++, --, +=, -=, /=, *=, <=, >=, ==, #=. The logic employed is: when a token contains a character that is in the special delimiter set (all characters of the operators above), the next token is

retrieved, if it's equal to '=' it means that we stand before an operator listed above, thus, we return the initial token plus the '=' as a single token. If both tokens are equal to '+' or '-', we also return both together. Otherwise, we now know that it is a regular token, therefore, we store the last token retrieved, to use it in the next token retrieval call, and we simply return the initial token.
The tokenizer also keeps track of the current location in the source code by incrementing the line count when a newline is found and incrementing the column count by the token size and ignored characters. Every time a newline is found the column count is also reset.

Back to the Lexical Analyzer, after retrieving the next token, the token type is determined by looking it up in the dictionary, which has all reserved words mapped with their type. If there is no match in the dictionary, the token is compared to 5 different regular expressions representing numeric literals, string literals, decimal literals, character literals and identifiers. If there is still no match, an error is raised to inform that the token provided is unknown.

Syntax Analyzer

This module, often referred to as *parser*, ensures that the sequence of tokens generated by the previous module is syntactically correct, meaning that it follows the grammatical rules of Z language. While doing so, it generates a parse tree representing the grammatical structure of the source code that will be then sent to the next module.

There are several types of parsers commonly used for compilers. Ours is a recursive parser, meaning that all the non-terminal productions of our grammar were translated to functions in this module that recursively call one another by following the grammar rules until they encounter a terminal. In this case, the *match* function is called to ensure that the current token is equal to the expected one. As our grammar is non-ambiguous (can only take one possible path), if we reach this matching point and the currently evaluated token is not equal to what we expect, we can be sure that the source code is not syntactically correct. Therefore, we try to perform error recovery and if this is still not enough, we raise a corresponding error and stop the execution of the program. This grammar trait makes our parser more efficient by avoiding backtracking (which happens when the wrong path is chosen).

As ours is a top-down parser, we start the process by taking the start symbol of our grammar as the root of our parse tree and proceed by expanding its non-terminals. To know which production to expand for those non-terminals that can have multiple representations, we use the *checkMatch* function. If the first production generated by the non-terminal being explored is another non-terminal, we send the current token and the *first* set of terminals in the expected non-terminal to the *checkMatch* function. On the contrary, if the first generated production is a terminal, we send the current token and the terminal itself. If the current token matches any of the other tokens sent as parameters, we know that is the right path to explore. However, if it doesn't, then we check the next production of the non-terminal.

Note that the *first* and *follow* set of productions from a non-terminal have been specified with a HashMap holding all the corresponding terminals. When calling the *checkMatch* function, if the *first* set of terminals contains *epsilon* (meaning it could be empty) then we append the *follow* set of the respective non-terminal to send to the *checkMatch* function.

The described technique is known as LL(1) parsing: left-most derivation with one symbol lookahead, as we start checking the left productions of a non-terminal, and only if they don't

match, we move to the next one (to the right). Moreover, we only use the current token to determine the path to follow (single lookahead).

The advantages of using a recursive parser instead of other methods such as the Table Parser are that this had a more straightforward implementation approach (translating the grammar productions to functions) and was easier to understand and visualize for everyone in the group. However, it also complicated the development process whenever any changes had to be made to the definition of our grammar, as applying these was quite tedious: we had to rewrite their functions and study their effect on the *first* and *follow* sets. The Table Parser method would have suppressed this problem by defining a dynamic way to traverse our grammar.

It is worth mentioning that it is in this stage that we decided to add the entries to the symbols table. In order to check whether or not to add an entry, we decided to check when we matched a token. Ideally, we would want to just add the entry when we find a name, however since we were filling the symbols table as we were creating the tree, if we only checked if the token was a name as a condition to add to the symbols table we would have been missing some information to add to an entry, therefore we had to tailor the conditions of adding to the symbols table to our specific grammar.

Lastly, we would also like to add on the error recovery concept we introduced in this module by explaining the mechanism we implemented to handle syntax errors and recover from them. As we previously mentioned, we use the *match* function to determine if the current token is equal to the one we expected. Whenever these two aren't the same, we check if the current token is equal to the next token we will receive. We achieve this by doing a new lookahead, asking for the next token to the Lexical Analyzer and comparing this with our expectations. If these match, we assume that the current token was never received and continue the execution as if the current token was the next one. This scenario would account for both unexpected tokens and repeated tokens, helping the user identify further errors by continuing with the execution of the program rather than terminating it.

Semantic Analyzer

The semantic analyser is last analyser of the compiler, and it checks that the parse tree generated by the syntax analyser is semantically correct, by traversing it and checking that the semantic rules, explained later, are followed. Assuring the next stage which is the intermediate code generator that the parse tree is semantically correct, so it can generate the intermediate code knowing that the parse tree is semantically correct.

The semantic analyse traverses the tree using an algorithm that mixes breadth first search depth first search. Using breadth first search to look for determined grammatical rules and when finding the key ones, it must check to see if the sematic is correct it then expands the sub-tree of that rule using depth first search, to look for the leaf nodes which are the ones containing the real value appearing in the code. It has different functions to explore the different relevant grammatical rules in a depth first search manner, but it had common function to check that two data types are equal or equivalent, the parameters of the functions follow the correct order and they are the same as in the declaration, check if the variable is initialized or declared before operating it or using it, check for duplicate names in the same scope, check that each function has at least one return statement that is going to be executed always and that it returns the appropriate datatype, etc. All these checks are made following the semantic section that appear later, where all the semantics are clearly explained.

In addition to the checks, when it finds an error, it must report it, to do so it uses the error handler to send it the semantic error or warning and then continues analysing the code. It does not stop after finding one error, it continues analysing the whole parse tree and then at the end it stops the compiler and prints all the errors. That way all the errors are properly detected and displayed for the user to see in one go, instead of having to compile and solve each problem one by one.

Finally, apart from checking that the tree is semantically correct it also sets some important attributes in the symbols table that are going to be needed later on and sets each variable node of the tree with its variable id, so the stage to come can get the information of a variable directly from the symbols table by using that id.

Intermediate Code Generator

This module tends to be the intermediate stage between the language and the machine. This stage receives a parse tree that is guaranteed to be linguistically correct and now needs to generate an output that will make it much easier to translate into assembly code. For our compiler, we have decided that the representation for this output be 3 address code (TAC).

Three address code is a representation for intermediate code generation in which the parse tree is assessed, and a list of entries in order are used to represent the different operations that need to be done in order. The structure used for our entries is described in the Own Structures section of our report. But for example, if our parse tree contains an if, the idea would be to generate entries to evaluate the condition, generate the entries for the body of the if and clearly indicate where it starts and ends. This way it is much easier to convert into assembly instead of directly translating the parse tree into assembly.

The 3 address code works by specifying which registers those operations should be done on, however in our case we haven't directly assigned an address for each register but rather a name, which would be 'r' followed by the register number. We have just made it so that every time a new variable or number is seen a register is allocated without worrying about the memory limit, we have done this so allowing the back end to manage the optimization of registers and thus decoupling our intermediate code generator from the machine.

Our intermediate code generator starts of by calling generateTacList() with the root node of the parse tree as a parameter, it has a big if checking for the grammar tag of the node, each case implements the TAC list required for certain sections of the grammar, if the grammar tag doesn't match any of the cases, the function is called recursively on the node's children, until all of the children have had a TAC list generated. This way the tree is traversed recursively, however it is not traversed from top to bottom, due to the fact that if a grammar tag matches in a case, a function is called to handle that case with the node as a parameter. Using the rules from our grammar, the tree is checked, and the suitable TAC list is created for that node and its children. It should be mentioned that in cases where the grammar tag is an if or something that can have a body of code inside, generateTacList() is called again as the code that can be in an if can also be found outside, and therefore it is already accounted for inside the big case in generateTacList().

Back-end

This second stage of the compiling process manages the translation of the received intermediate code to target machine code. This stage usually includes an Optimizer module to improve the efficiency of the code considering the target's functionalities, however, due to time constraints, our implementation skipped this section and focused mainly on machine code generation.

Target Code Generator

Until now, modules were fully decoupled from the target machine, meaning that they could all be reused to generate a different output code. This module, however, is entirely implemented based on the selected machine code and thus, does not offer output portability. The machine code generated follows the MIPS[1] I instruction set architecture which is a load/store or register-register architecture. This describes architectures where instructions are divided into two categories: memory access and ALU[2] operations, which take place only between registers.

The MIPS I architecture has 32 GPR[3] of 32-bit size:

- $zero: holds constant 0
- $at: reserved for assembler
- $vo-$v1: expression evaluation or procedure return values
- $a0…$a3: procedure arguments
- $t0…$t9: temporary registers (not preserved across calls)
- $s0…$s7: saved temporary registers (preserved across calls)
- $k0-$k1: reserved for OS kernel
- $gp: pointer to global area
- $sp: stack pointer
- $fp: frame pointer
- $ra: procedure return address

Our generated code uses all temporary, stack pointer, frame pointer, return address and return value registers. To allow floating-point operations, the architecture has a specific coprocessor for them named Coprocessor 1. This coprocessor can only operate on the 32 specific floating-point registers available ($f0-$f31), these registers are paired for double precision values restricting the use of the odd numbered ones, therefore, there are only 16 usable registers which our code generation uses.

[1] MIPS: Microprocessor without Interlocked Pipelined Stages

[2] ALU: Arithmetic Logic Unit

[3] GPR: General Purpose Register

The process followed to generate code is specific to each operation, the list of TAC entries is traversed, calling for each entry the procedure associated to the operation to fulfil. The TAC entries reference registers, internally named TAC registers, in an unlimited manner with the format “r3”, for instance. It is the target code generator’s task to map the TAC registers to the limited temporary registers available and persisting them to the memory stack when required.

DECLARE

Whenever a DECLARE entry is received, we first check whether we are currently in a global context or not, in order to determine whether we must make a global or a local declaration. The global context disappears whenever a function or CEO is declared. If it’s a global declaration, we reserve space for it in a general memory area and we create a new register association with the received label. Otherwise, we reserve space in the stack, in case it must be persisted in the future, by moving the stack pointer down and mapping the TAC register with the stack pointer’s frame pointer offset. For both cases, if the declaration is for a floating-point variable, the TAC register is marked as floating-point, in order to properly handle future operations with it, as floating-point operations must be done with specific instructions and in their specific registers.

LITERAL

To store a literal, a temporary register is assigned, choosing from a regular register or a floating-point register, depending on the value. Then, the literal is loaded to the register.

The process to assign a temporary register, which is frequently executed due to their limited amount, involves looking first whether there is already a register assigned to the TAC register received, if this is the case, the assigned register is returned. Otherwise, a register is retrieved from a queue (where the register retrieved is the one that has been assigned longer) persisting its value (to global memory or stack) and removing its TAC - temporary association. The new TAC - temporary register association is then stored, and any persisted value is loaded.

EQUAL

Once registers have been assigned to the source and destination TAC registers, 4 cases must be handled differently, as regular registers are not directly compatible with floating-point registers:

- Floating-point destination and floating-point source
- Floating-point destination and regular source
- Regular destination and floating-point source
- Regular destination and regular source

Those directly incompatible cases require converting integer to floating-point or vice versa and moving to/from Coprocessor 1.

NEG

Receiving also a source and destination, the exact same steps as EQUAL are followed with the additional instruction to negate.

ADD

As usual, temporary registers are associated to the destination and both operands. To provide a general and simple course of action, if an operand holds a floating-point value, the addition is done in the Coprocessor 1, in other words, the operation is done with precision. If the other operand is not in a floating-point register, an auxiliar register is used to move it to Coprocessor 1.

SUB

The same exact steps as ADD are executed for the subtraction of registers with, of course, the regular and floating-point subtraction instructions instead.

MUL

The same exact steps as ADD are executed for the multiplication of registers with, of course, the regular and floating-point multiplication instructions instead.

DIV

The same exact steps as ADD are executed for the division of registers with, of course, the regular and floating-point division instructions instead. The division between two regular registers stores the result and remainder in the reserved “hi” and “lo” registers, the result is moved from them to destination using an additional instruction.

MOD

The same steps as DIV are executed for the modulo of registers but, for floating-point values the manual calculation of the modulo is required using the division instruction, as there is nothing as the remainder register "hi" in Coprocessor 1:

$$DIV = \frac{operand1}{operand2}$$

$$MOD = operand2 \cdot (DIV - |DIV|)$$

AND

As our semantics guarantees that AND operations are only allowed between Booleans, we don't need to handle operations between floating-point values. Thus, we simply associate registers to the destination and the two operands, then, we append the AND instruction.

OR

The OR operation is done exactly as AND but with the OR instruction.

COMPARISON

The COMPARISON entry compares two registers as: equal, not equal, less than or equal, greater than or equal, less than and greater than. As comparisons between floating-point values are allowed, to ease the implementation and define a general format, all comparison operations occur in Coprocessor 1.

After moving the values to Coprocessor 1 using auxiliar registers, depending on the comparison to perform, an instruction is executed out of the floating-point instructions *compare if equal, compare if less than or equal* and *compare if less than*. Opposite comparisons such as "compare if greater than" are achieved by simply swapping the registers. Then, an instruction that moves a value from a register to another if the last condition was true, moves a "1" to a register initially loaded with a 0. The "not equal" comparison which uses instruction *compare if equal* moves it if the condition is false instead. The result is finally moved to the destination register.

STARTER

The STARTER entry mainly indicates the start of a statement body (if, else if, else, while, do while and for) but also indicates the code generator the starting point of entries required to compute the condition of a statement, as they must be handled differently considering they may be required to execute more than once (while, for instance) or at an earlier/later point (if, else if... for instance).

- **Conditions**

  When a STARTER entry with the tag "conditions" is processed, a flag is enabled and a list is initialized. The flag indicates the TAC entry traversing loop to store the next TAC entries to the list until another entry of type STARTER or ENDER is found (actual start/end of the statement). This allows storing the entries related to a statement condition to be processed later (see *Ender*).

- **If**

  As the next TAC entries will represent code inside the If's body, we want to branch to first check the If's condition and if it evaluates to true return to this point to execute the If's body.

  To perform this, we first want to clean our context which involves persisting all registers and removing all register associations. This is done to avoid using inside the If body associated registers which may have used for other associations during the condition checking process. The generator is reading code in a top-down manner, but the assembler will branch moving between contexts, hence the need of context cleaning.

  Then, we can add a GOTO instruction to branch using a label to where the condition checking process will be (added later, see *Ender*) and add a label to allow returning if the condition is true. The labels used make use of a unique identifier to allow nested statements, in other words, statements inside statements.

  We then initialise a list and add it to a Map with the previously mentioned identifier as the key, this list will contain the check instructions with the corresponding TAC register that will hold the result of the statement condition. This list, which will be processed later (see *Ender*) allows concatenating as many Else If as the user wants. The identifier is then pushed to a stack that enables the use of nested statements.

- **Else If**

  When an Else If is encountered, the same logic as an If is executed except for adding the GOTO instruction, as the only way of reaching this point will be by branching from the condition checking process (the end of the previous If/Else If body already has a GOTO instruction to avoid executing different bodies). The check instructions list is updated by retrieving the first identifier in the statement stack and the associated list in the check instruction Map.

- **Else**

  The same exact logic as Else If is followed when an Else is encountered.

- **While**

  The same exact logic as If is followed when a While is encountered.

- **Do While**

  The same exact logic as While is followed when a Do While is encountered except for adding the GOTO instruction, as the body must be executed at least once, we don't want to check the condition yet.

- **For**

  The same exact logic as While is followed when a For is encountered.

ENDER

The ENDER entry is processed when the end of a statement body is reached. We first obtain the statement's identifier from the statement stack. If the statement is of type If, Else If or Else, we clean the context and add a GOTO instruction to the end of the whole statement processing, in other words, to the beginning of what is after the statement in the source code. This allows continuing with the execution once the body of the statement has been executed, without executing the condition checking process again which may be just below.

The TAC entry also carries a tag that indicates if the statement chain has another statement next, for instance, if there is an Else If after ending the If body. If there is another statement, we don't want to check the condition yet, thus, we don't do anything more and allow the process of the next entries. Otherwise, we can proceed with the condition checking process and the branching cases.

We first want to clean the context and add the label we initially defined in the corresponding STARTER. We then retrieve the check instructions list and the condition-related entries for each check instruction. After processing all entries, the condition registers of the check instructions are evaluated with the *branch if greater than* instruction, or a simple GOTO is executed in the case of Else, as there is no condition to check. Finally, the statement stack is popped and a "continue" label is added to allow the execution of the code after the execution of the statement.

FUNC_DEC

This entry is created when a function is declared. If we are in a global context, it means that we've finished declaring global variables, thus, the global context flag is cleared and a GOTO instruction to the CEO procedure is added to start the code's execution. A label is placed to allow "calling" the function later and the return address is stored to stack to allow nested function calls by reusing the return address register. This is done here and not on function call because we use the instruction "jal" which jumps to the function and stores the address to return to, we don't know the return address until the instruction is executed and after executing it, we are now here, in the function's body. There are other ways of retrieving the return address, but this is a simple way of handling it. Finally, the stack pointer's frame pointer offset is reset for the new context.

PARAM_DEC

A PARAM_DEC entry is created for each parameter in a function declaration (after FUNC_DEC). We simply associate its stack space to the TAC register received. Parameters are declared (on function declaration) and sent (on function call) in the same order, thus, the sequential stack space allocated for each is the same in both cases.

PARAM

PARAM entries indicate the parameters to send (and store to the new stack context) before a function CALL entry. The entries carry the TAC register holding the parameter value, these registers are simply stored to a list to handle them later (See *Call*).

CALL

The CALL entry indicates the calling of an already declared function, it holds the function's name or label as entry argument. As we're moving to another area, we first want to persist all registers. Then, the frame pointer is stored to the stack to allow returning to the current frame context later, additional space is also reserved for the return address (stored on function execution) and finally, the frame pointer is set to the value of the stack pointer, creating a new frame context.

At this point, in the new context, parameters are placed in the stack to allow their access inside the function's body. Lastly, the *jump and store return address* instruction is added. All register associations (jump to new context) and the previously mentioned parameter list are also cleared.

RETURN

The RETURN entry indicates the end of a function's body and thus, the return to the original context. The stack pointer is set to the value of the frame pointer and moved up two positions (original frame pointer + return address), bringing it back to the original stack pointer position which is the end of the original context. The value right above the frame pointer is moved to the return address register and the value above this one is moved to the frame pointer, setting the frame pointer to the original frame pointer value. Finally, the return address register is used to jump back to the original context. All register associations are cleared as we are moving to another context.

Grammar

A grammar is a set of instructions that indicates how to write valid sentences (statements) in Z language. Its productions enable the creation of parsers that convert source code into a tree-shaped representation of its syntactic structure. This parse tree can then be utilised for analysing or processing the source text further.

The definition of our grammar is deterministic and non-ambiguous, meaning there is always only one possible correct path to follow (there will never be two productions from the same non-terminal that produce the same terminal as a first result). Moreover, we avoided the issue of left recursion by ensuring no non-terminal can have itself as the first production of one of its paths (as this could be infinitely expanding itself).

Below you can find the list of productions of our grammar expressed in BNF format and divided in modules.

Start

<start> ::= <globals> <main>
<globals> ::= <globals_decl> <globals> | E
<globals_decl> ::= <constant> <declaration>

Declarations

<declaration> ::= <arr_decl> | zombie name <func_decl> | <datatype> name <func_or_var_decl>
<func_or_var_decl> ::= <var_decl_assign> | <func_decl>
<constant> ::= fact | E

<var_decl_assign> ::= = <var_op> . | <equal_assign_op> <var_op> . | .
<var_op> ::= <neg> <var_op_val>
<var_op_val> ::= <var_value> <nested_op> | (<var_op>)
<nested_op> ::= <operator> < var_op_val > | E
<operator> ::= + | - | <high_priority_operator>
<high_priority_operator> ::= * | / | %

<arr_decl> ::= fam <arr_dim> <datatype> name <arr_assign> .
<arr_dim> ::= numeric_literal <arr_arr>
<arr_arr> ::= <arr_dim> | E

<arr_assign> ::= = <arr_assign_val> | E
<arr_assign_val> ::= [<arr_list>] | <var_value>

<arr_list> ::= <arr_value> <arr_value_list>
<arr_value> ::= <neg> <var_value> | [<arr_list>]
<arr_value_list> ::= , <arr_list> | E
<var_value> ::= <literal> | name <arr_or_func>
<arr_or_func> ::= <arr_pos> | ¿ <value_list> ? | E
<arr_pos> ::= [<var_value>]
<neg> ::= # | E

<datatype> ::= bro | sis | twin | bipolar | mainchar
<literal> ::= <comparable_literal> | <non_comparable_literal>
<non_comparable_literal> ::= str_lit | null_lit
<comparable_literal> ::= numeric_literal | decimal_literal | boolean_literal | char_literal

Function Declarations

<main_> ::= CEO ¡ <func_body> !
<func_decl> ::= ¿ <func_param> ? <func_impl>
<func_impl> ::= ¡ <func_body> ! | .

<func_param> ::= <param_decl> | E
<param_decl> ::= <datatype> name <param_list>
<param_list> ::= , <param_decl> | E

Function Body

<func_body> ::= <expression> <func_body> | <end> | E
<expression> ::= <conditional> | <while_loop> | <for_loop> | <switch_> | name <func_or_var>
| <datatype> name <var_decl_assign> | <arr_decl>
<func_or_var> ::= <func_call> | <var_assign> | <arr_pos> <arr_assign>
<end> ::= <return> | E
<return> ::= throwback <return_value> .
<return_value> ::= <neg> <var_value>

Function Call

<func_call> ::= ¿ <value_list> ?.
<value_list> ::= <var_op> <var_list_more> | E
<var_list_more> ::= , <value_list> | E

Variable Assignation

<var_assign> ::= <assignation> .
<assignation> ::= = <var_op> | <equal_assign_op> <var_op>
<equal_assign_op> ::= += | -= | *= | /=

While Loop

<while_loop> ::= vibe <while_opt>
<while_opt>::= <while_> | <do_while>
<while_> ::= check ¿ <boolean_cond> ? ¡ <func_body> !
<do_while> ::= ¡ <func_body> ! then check ¿ <boolean_cond> ?

Conditional Statement

<conditional> ::= <if_> <cond_else>
<if_> ::= like ¿ <boolean_cond> ? ¡ <func_body> !
<cond_else> ::= whatever <else_opt> | E
<else_opt> ::= <elif> | <else_>
<elif> ::= like ¿ <boolean_cond> ? ¡ <func_body> ! <cond_else>
<else_> ::= ¡ <func_body> !

<boolean_cond> ::= <neg> <condition> <nest_cond>
<condition>::=name <comparison> | <comparable_literal> <comparison>| (<condition>)
<nest_cond> ::= and <boolean_cond> | or <boolean_cond> | E

<comparison> ::= <comparison_op> <var_op> | E
<comparison_op> ::= < | <= | > | >= | == | #=

Switch Statement

<switch_> ::= swipe ¿ name ? ¡ <case_> <default_> !

<case_> ::= right <literal> : <case_body> <nested_case>
<nested_case> ::= <case_> | E
<default_> ::= left : <case_body> | E
<case_body> ::= <func_body> <break_> | E
<break_> ::= periodt . | E

For Loop

<for_loop> ::= 4 ¿ <for_decl> <boolean_cond>. <for_iterator> ? ¡ <func_body> !
<for_decl>::= <for_var_type> name <var_decl_assign>
<for_var_type> ::= <datatype> | E

<for_iterator> ::= name <it_change>
<it_change> ::= ++ | -- | <equal_assign_op> <var_value>

Language Specifications

As previously described, Z is a **high-level** programming language that aims to ease the coding experience for Gen Z programmers in all possible fields. Hence, it is designed as **a general-purpose** language to allow coding in a wide variety of application domains, without specifically targeting any industry.

Rooting from the C family, Z shares some similarities with this well-known and widely used language. Z is a **strongly** and **statically typed** language, meaning that all constants and variables must be assigned a particular datatype and will be bound to it during compilation time.

Regarding its syntax, Z follows the conventions dictated by the C language by using symbols to indicate the termination of statements (.), to denote blocks of code (¡!), or to indicate condition statements (¿?). These symbols might be different from the ones commonly used by many programming languages, but they mean to facilitate the readability of the code by more accurately matching their purpose.

Z is a language offering a **procedural** paradigm. By being procedural, Z allows its users to specify each step required to solve a problem, giving them maximum control over their code. These steps will be then grouped in procedures or functions encouraging its users to have a structured code with functionalities separated in different blocks.

Dictionary

This section contains a list of all the lexical belonging to the Z language, including its translation from the C language as well as a short description of use for the fields that require clarification.

Datatypes

C	Z	Overview
int	bro	datatypes that refer to numerical values are represented with actual members of the family.
float	sis	
boolean	bipolar	bipolar is mental illness that causes unusual shifts in a person's mood. Ranging from extreme highs (1) to lows (0). Precisely as booleans.
char	mainchar	as the name indicates, it is a nod to the datatype char.
[]	fam	arrays will be defined by specifying its datatype followed by the *fam* keyword. Ex. *bipolar fam 8*.
void	zombie	zombie represents sense of emptiness or lack of purpose.

Literals

C	Z	Overview
true	smash	based on the Gen Z app Tinder, `smash` is when you fancy someone (`true`) and `pass` the opposite (`false`)
false	pass	
null	ghosted	the practice of suddenly avoiding all communication. When texting them, there is no response (null).

Delimiters

C	Z	Overview
(	¿	¿? reproducing a question, what we want to find out.
)	?	
{	¡	¡! is the answer to our troubles (texting style)
}	!	
;	.	. used to indicate the end of a sentence (;).

Arithmetical Operators

C	Z
+	+
-	-
*	*
/	/
%	%
++	++
--	--
+=	+=
-=	-=
*=	*=
/=	/=

Logical Operators

C	Z
&&	&
||	|
>	>
<	<
>=	>=
<=	<=
==	==
=	=
!=	#=
!	#

Reserved Words

C	Z	Overview
return	throwback	When wanting to `return` something we `throw it back`
const	fact	`fact` when something is known to be true (`const`)
break	periodt	`periodt` is used to signal the end of a discussion, similar to `break`.
main	CEO	`CEO` is used to indicate that someone is good at something.

Conditional Statements

C	Z	Overview
if	like	`like` is used as filler word to start a sentence, simulating an `if`. `Whatever` is used to suggest that we do not care about the outcome (`else`). `else if` is therefore used as `whatever like`.
else	whatever	
switch	swipe	Based on the Gen Z app Tinder, `swipe` (`switch`) can end up with different results that can be generalized into right or left. Being `right if` the positive aftermath and `left` our default one.
case	right	
default	left	

Loops Statements

C	Z	Overview
for	4	due to its similarity when pronouncing.
while	vibe check	passing a vibe means being authentic, hence we will only stay in the loop when x conditions are meant resulting in genuine interactions.
do	vibe	
while	then check	

Comments

C	Z	Overview
/*	<<	Comments in Z try to replicate the regular texting style for annotations in messages.
*/	>>	

Compilation Results

C	Z	Overview
syntax error	canceled	in Z language, you directly get `canceled` if a typo is made.
lexical error	cringe	Having lexical errors is `cringe`.
semantic error	boomer	if our compiler can't even understand your written code, you must be a `boomer` .
linker error	bombastic side eye	if we can't even generate the executable file, you do deserve a `bombastic side eye`.
success	stan	in Z, we `stan` for codes that work.
warning	sus	when something is `sus`picious (`warning`).

RegEx

Regular expressions are key in compilers as they enable efficient tokenization, pattern matching, error detection, whitespace handling, optimization, and adherence to Z language specifications during the compilation process. Hence, they are essential for accurately analyzing and processing the source code of programming languages.

Datatypes

Z	Regex
bro	^bro$
sis	^sis$
twin	^twin$
bipolar	^bipolar$
mainchar	^mainchar$
fam	^fam$
zombie	^zombie$

Literals

Z	Regex
numeric_literal	^-?[0-9]+$
decimal_literal	^[+-]?([0-9]*[’])?[0-9]+$
boolean_literal	^(pass|smash)$
char_literal	^'[a-zA-Z0-9.*]?'$
string_literal	^"[a-zA-Z0-9.*]*"$

Reserved Words

Z	Regex
throwback	^throwback$
fact	^fact$
periodt	^periodt$
CEO	^CEO$

Conditional Statements

Z	Regex
like	^like$
whatever	^whatever$
swipe	^swipe$
right	^right$
left	^left$

Loops Statements

Z	Regex
4	^4$
vibe	^vibe$
check	^check$
then	^then$

Delimiters

Z	Regex
¿	^¿$
?	^?$
¡	^¡$
!	^!$
.	^.$
[	^[$
]	^] $
,	^,$

Arithmetical Operators

Z	Regex
+	^+$
-	^-$
*	^*$
/	^/$
%	^%$

Logical Operators

Z	Regex
and	^&$
or	^|$
>	^>$
<	^<$
=	^=$
not	^#$

Variable/function names

Z	Regex
name	^[a-zA-Z]+[a-zA-Z0-9_]*$

Semantics

Semantics refers to the meaning conveyed by the elements and structure of Z language. The importance of semantics in a compiler lies in its role in ensuring correctness, understanding the source code, enabling optimizations, detecting errors, supporting language features, promoting portability, and complying with language standards.

1. **Variables**: All variables must be declared with a specific data type and name, and it can only store values of that datatype. Once declared, variables cannot change their data type. In addition, variables can be declared at the beginning of the code, at the beginning of a function or at the beginning of a block. Depending on where they are declared they will have different scopes. When declared at the beginning of the code they will have a global scope, at the beginning of a function their scope will be that function and at the beginning of a block their scope will be the block. Note that the scope of a variable declared in a function or block extends through all the function/block from the point where it is declared onwards. Also, the name assigned to a variable must be unique in its scope(note that a variable with the same name cannot exist in the scope of a block if the function where that block is has a variable with that name already declared), variables with the same name can coexist if their scope is different, but in those cases, the variable declared in a more "inner" scope will shadow the other/s. Constant variables (fact) once defined cannot be modified during runtime. Moreover, all variables are static, meaning that they have a fixed size and cannot be changed. All the basic types will have a fixed size which cannot be changed: bro's, sis's, bipolar's and mainchar will always have a fixed size of 32 bits. Note that variables and arrays must be declared, but they do not need to be initialize, this can be done later, also they can only be declared once.

```
fact mainchar letter = 'A'
```
1.1 Declaration of a constant

```
bro average = 3.
```
1.2 Declaration of a variable

When declaring an array of a datatype (fam), the size can be specified. But once set, it cannot be changed. Moreover, when making arrays of mainchar's, note that an extra space will be needed for the null character (\0), which is used to recognize the end of a mainchar array (String).

```
fam 3 sis arr = [ 2.9, 5.8, 8.0 ].
```
1.3 Declaration of a fam

2. **Function declaration**: All functions must be declared with a specific return type, a name, its parameters declaration and the body of the function. The throwback type, as with variable type, cannot be changed and the throwback must return a literal of the specified type. In addition, function name must be unique and follows the same rules as variables, there cannot be two equal names defined in the same scope, be it 2 variables, 2 functions or 1 variable or 1 function. Functions are always declared in the global scope and are always globally visible. Therefore, there cannot be a global variable with the same name as a function, but there can be local variables (block or function scope) with the same name as a function. Moreover, and as said with the variables, their scope extends from their declaration onwards, so a function cannot call the function/s declared above. Even though, it is possible to forward references, meaning that it is possible to declare a function above with its name, throwback type and parameters, but instead of opening "¿?", it ends with a "." and the function is declared below with its body. In addition, each program will be executed from the CEO function. Therefore, each program will need to have a CEO function. In addition, this function(CEO) will not have a throwback type nor will it have parameters.

```
bro functionName ¿ bro var ? ¡
        throwback 1.
!
```

2.1 Declaration of a function

```
CEO ¡

!
```

2.2 Declaration of CEO function

3. **Syntax**:
   - Termination of statements: ".".
   - Block initiation and termination: "¡" and "!".
   - "¿" and "?" are used for enclosing: conditional statements, function parameters declaration.
   - "(" and ")" are used as in math's for arithmetic operations, giving priority to the operation enclosed by those symbols.
   - Decimal points: " ' ".

4. **Statements**:
   - Expression statements.
   - Conditional/selection statements (like and swipe).
   - Iteration statements (4, vibe check, vibe-then check).
   - Jump statements (throwback, periodt)
   - Labeled statements (right and left statements used withing a swipe statement)

   a. Expression statements: An expression statement consists of an expression followed by a ".". The execution of such a statement causes the associated expression to be evaluated. When executing a expression they are evaluated. Each expression whose value is 0 will be evaluated as false and the rest as true. Considering value the value they return in the case of a logical expression or the byte value they form.

```
total += arr[i].
```

4.1 Adding value of position i of farm arr to value of average variable

   b. like statements: It is a selection statement which has three forms:
      i. like ¿ expression ? ¡ block !
      ii. like ¿ expression ? ¡ block ! whatever ¡ block !
      iii. like ¿ expression ? ¡block! like whatever ¿ expression ? ¡ block ! whatever ¡block!

   Like statement works as follows: if the expression inside is evaluated as true then then the block is executed, otherwise it is not. If the expression is evaluated as false and there is a whatever clause is present, then the block of the whatever is executed. If the like expression is evaluated as false and there is a like whatever then its expression will be evaluated, if is true it will execute its block and if it is false, then it will evaluate the next like whatever, if there is, or execute the whatever block, if there is.

```
like ¿ average >= 9 ? ¡
        result = "You pass with honors"
! whatever like ¿ average >= 5 ? ¡
        result = "You pass"
! whatever ¡
        result = "You do not pass"
!
```

4.2 Third form of like statement

c. Swipe statements: It is a selection statement with the following form: swipe ¿ expression ? ¡ body !
Where the body is a composition of right labeled statements and a left labeled statement. When executing a swipe statement, the expression is evaluated, and the control is transferred to the right labeled statement whose labeled value matches the value of the expression. If no right statement has the same value, then control is transferred to the left statement, if there is one. Otherwise the control is transferred to the next statement after the swipe. Note that if the labeled statement does not include a periodt at the end of its body, then all the following labeled statements after the executed one will also be executed regardless of whether the value corresponds to the one of the expression or not.

```
swipe ¿ num ? ¡
!
```

4.3 Swipe example

d. A vibe check statement is an iteration statement and is formed by the following sequence: vibe check ¿ expression ? ¡ block ! . A vibe check statement evaluates the expression, if it is evaluated as true, then the block is executed and the expression is evaluated again. If the expression is evaluated as false then the next statement after the vibe check is executed. Therefore, the vibe check statement will keep executing the block as long as the expression is true.

4.4 Vibe check example

e. A vibe – then check statement is an iteration statement and is formed by the following sequence: vibe ¡ block ! then check ¿ expression ? . This statement works as a vibe check statement, but the difference is that it will always execute the block first and then evaluate the expression. Therefore, the block will be executed at least once.

4.4 Vibe then check example

f. 4 statement is an iteration statement and is formed by the following sequence: 4 ¿ initializer. expression. update ? ¡ body ! . Initializer is a variable declaration, be it either a declaration or a variable setting. Then the expression and at the end the update expression. When a 4 statement is executed, the initializer expression is evaluated and then the expression is evaluated. If the expression is evaluate as true, then the body is executed and then the updated expression is evaluated. Then, as the vibe check the expression is evaluated again. Note that the initializer expression is only evaluated only the first time and that the update expression is evaluated always after executing the block. This statement will continuously execute the block and update expression as long as the expression is evaluated as true. Also, neither initializer, expression or update can be omitted when declaring a 4 statement.

```
4 ¿ bro i = 0. i < 10. i++ ? ¡
        total += arr[i].
!
```

4.5 4 example

g. Throwback statement, this is a jump statement, which means that when evaluated jumps from one part of the code to another. The throwback statement must appear within the body of a function followed by a value (be it literal or a variable storing the value). The value must be of the same type as the throwback type stated in the function declaration. The throwback indicates that the current function execution has ended, and it will return the value to the part of the code where the function was called, substituting the function call by the value returned by the throwback and evaluating that statement with the value returned.

```
throwback 0.
```

4.6 thrwoback statement

h. Periodt is the other jump statement in Z language which may appear withing iterative statements or a swipe statement. It indicates that the statement being executed must end. Given control to the next statement after the one where the periodt is.

i. A labeled statement can either be right or left and they can only appear withing the body of a swipe statement. Right must be followed by a constant-expression and a body and left appears together with a body. Labeled statements identify the targets for control transfer in the swipe statement. Giving control to the body of the labeled statement whose constant expression matches the expression result of the swipe statement. As said in the swipe statement, if no periodt appears within the bodies of labeled statements. Then, after executing the body of the labeled statement to whom the swipe gave control to, it will continue executing the bodies of the following labeled statements if they exist.

```
swipe ¿ num ? ¡
        right 0:
                throwback -1.
        right 1:
                throwback 0.
        left:
                throwback num*num.
!
```

4.7 swipe example

5. **Comments**: The code might be commented, to do so the symbol “<<” must be used before the comment. From the symbol “<<” onwards, all lines will be treated as comments until the symbol “>>” appears, where the lines will stop being treated as comments.

6. **Mathematic Expressions:** There are different mathematic operations supported by Z (addition, subtraction, multiplication, division and module(given 2 numbers, its result is the reminder of the division of the first number over the second)).
    a. **Priorities:** They follow the same rules as in math. Therefore, multiplication, division and module have the same priority which is higher than the one for addition and subtraction, which also have the same level of priority. In addition, as in math’s, parentheses gives priority to the operations inside it (in case of nested parentheses, the one inside another parentheses is the one with higher priority) and the operations are evaluated from left to right. Therefore, given the operation 4 * 4 / 2: the result will be 8 as multiplication and division have the same level of priority, so the leftmost operation (multiplication) is evaluated first and is result is then divided by 2.

b. **Assignment operations:** Apart from the normal symbols (+, -, /, %), the next symbols can also be used (+=, -=, *= and /=). These take the value stored in the variable at the left of the symbol and the value expressed at the right and perform the operation represented by the first symbol next to the equal and then store it in the variable at the left of it.

```
bro result = 3.
result *= 4.
```
6.1 operation with a variable

Given the code above, 3 is stored in result and then with the symbols *=; the value in result (3) is multiplied by 4 and then the result (12) is stored in result.

c. **Post-increment/decrement** also exist: ++ and --. These symbols must be used after a variable(`result++.`) and will add/subtract 1 to the value stored in the variable and the result will be stored in the variable.

d. **Cross datatype operations:** Arithmetic operations can be performed between different datatypes. When performing operations between numerical datatypes (bro, sis and twin) twin has the most priority followed by sis. Meaning that when performing operations between twin and sis or bro, the result will always be of type twin. And when performing operations between sis and bro the result will be of type sis. Cross datatype operations will always follow this hierarchy. The same way operations can be made with mainchar values. In those cases the mainchar will take the ASCII value of the character being operated and it works as bro when operated with sis and twin(mainchar + twin = twin and mainchar + sis = sis). But when operated with bro, if the result is going to be stored in a variable of type bro or going to be treated as a bro it will be stored as a bro (32 bits). But if it is going to be stored or treated as a mainchar, then only the first 8 bits will be taken.

7. **Logical Expressions:** There are different logical operators in Z(|, &, >, <, >=, <=, ==, #=, #), which can be used to form different logical expressions. In addition, the use of parentheses may be used as in math's to give priority to the expression/s encapsulated by them. Otherwise, the expression will follow its normal priority rules and in case of a tie from left to right. >, < , >=, <=, ==, #= and # are used to convert from statement expressions to 0 or 1, while | and & are used to evaluated if different combinations of expressions evaluated as 0 or 1 return 0 or 1. Therefore, the first have the same priority and will be evaluated before evaluating any | or & expression. Then, between | and &, & has a higher priority than |. Therefore, given the expression: a & b | c, it will be evaluated as the expression: (a & b) | c, meaning that the logical expression with the & will be evaluated first, and then the result of that expression will be evaluated with the operator | and c.

    a. **Logical OR expression:** Logical OR expressions are formed by 2 expression statements or logical expressions with an OR operator ("|") in between.

    expr1 | expr2

    When evaluating an OR expression, expr1 will be evaluated first. If it is non-zero, then the value of the parent expression is 1. Otherwise, the value of the parent expression is the value of expr2.

    b. **Logical AND expression:** Logical AND expressions are formed by 2 expression statements or logical expressions with an AND operator ("&") in between.

    expr1 & expr2

    When evaluating an AND expression, expr1 will be evaluated first. If it is zero, then the value of the parent expression is 0. Otherwise, the value of the parent expression is the value of expr2.

c. **Comparison expression:** Operators: >, <, >=, <=, ==, #= are used to form comparison expressions. They are formed by an expression, one of the operators previously stated and another expression.

expr1 > expr2

Operators > check that the value of expression 1 is greater than the value of expression2. On the other hand, operator < checks that the value of expression 1 is smaller than the value of expression 2, or what is the same that the value of expression 2 is greater than the value of expression 1. They are evaluated as True (1) or False (0) depending on whether they comply with the operator rules or not. Operators >= and <= work the same as > and < but apart from checking if x value is greater or smaller than, they also check if the value is equal. Therefore, they work the same as > and <, but they include the case where both expressions are equal as True (1).

Operator == checks whether expression 1 is equal to expression 2, evaluated as 1 if they are equal and to 0 if they are not equal. Operator #= works the other way, evaluated as 1 if they are not equal and 0 if they are.

d. **Negation expression:** Negation expressions are formed by the operator # and an expression, and it works as a nor gate. It is formed by the operator # followed by an expression.

#expr1

Expr1 must be a variable of type bipolar or a logical expression. After the negation symbol, if there are parentheses, then the resultant value of evaluating the expressions inside will be negated. If there are not, then the expression following the symbol will be negated.

Work phases

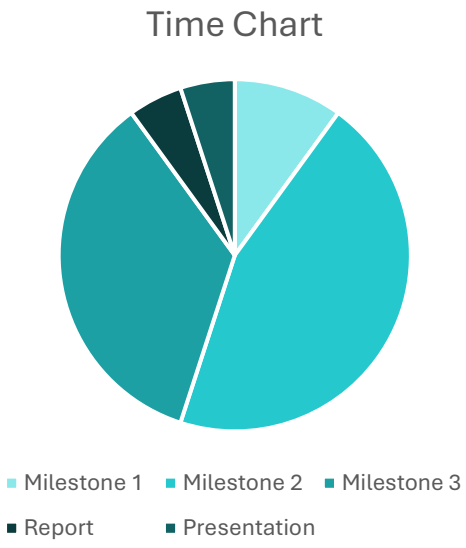
To effectively manage our project, we used Bitbucket and Jira. Bitbucket, which uses GIT, functioned as our code management system, while Jira facilitated the creation of tasks and planning of our sprints.

For a successful implementation, we adopted a progressive debugging approach for each compiler function as it was developed. This strategy allowed us to address issues incrementally. Consequently, our milestones were structured as follows: the first milestone focused on language design, the second on developing the pre-compiler, lexical analyzer, and syntax analyzer, and the third on the semantic analyzer, intermediate code generator, and target code generator. Although the workload was substantial and it was not always possible to thoroughly test each component by the milestone class deadline, we ensured that by the end of each sprint, we had met our objectives, and everything worked perfectly.

- Language design.

- Pre-compiler
- Lexical analyzer
- Syntax analyzer.

- Semantic analyzer.
- Intermediate code generator.
- Target code generator.



Own structures

In order to implement all of the above-mentioned modules of our Z compiler, we had to create our own data structures to store and pass the necessary information between the different sections. We will now detail the design and functioning of each one of these.

Token

We created a *Token* class to represent all the input symbols identified by the lexical analyzer module. This included the following two attributes:

```
private final TokenType type;
private final String value;
```

The real value of the token scanned from the source file is stored in the *value* attribute, while the type of the token is stored in *type*. The latter is determined by the lexical analyzer with the help of our dictionary, which we will now explain.

Dictionary

```
private final Map<String, Token.TokenType> dict;
```

The way we chose to represent our language's dictionary was with a *HashMap* for easy and fast retrieval. This one was populated at the beginning of the execution of our compiler with all the reserved symbols from Z as its keys and their respective *TokenType* as its value. This *TokenType* is an enumeration that can take any of the following values: *KEYWORD* ("fact", "fam", "zombie" ...), *SEPARATOR* (".", "(", "!" ...), *OPERATOR* ("+", "=", "and" ...), *NULL_LIT* ("ghosted"), and *BOOL_LIT* ("smash", "pass").

This process was especially useful for the lexical analyzer module to identify and classify the received input tokens. The process involved checking if the dictionary had a value for the input key. If this was the case, we had already identified the token. On the contrary, we had to check if the token matched any of the predefined regex (listed in the *Language Specification* section) to find the remaining *TokenType* values: *NUM_LIT* (2, -4), *STR_LIT* ("hello", "TX10"), *DEC_LIT* (4´5, 10´0), *CHAR_LIT* ('a', 'T'), *NAME* (var1, getFibonacci). In the case of not being able to match the input token to any of these types, we raise an *UnknownTokenError*.

Symbols Table

Our symbol table aims to implement two main functions, the lookup and the insert with the minimum complexity cost possible, in our case both the insert and lookup are O(1). In order to achieve this, we have made the symbol table to be 1 hash map, and since we want to allow the different modules to be able to perform the lookup with the information available in each node in the parse tree, we had to make the access key for the hash map a string which would be the name. Since variables can have the same name but be in different scopes, we decided that the key remains as the name, but the value to be a list of SymbolRows (since there are likely not many variables with the same name compared to total variables there are, this does not raise the time complexity of the operations of the symbol table) where each SymbolRow would have a unique id. Therefore, making the symbol table the following:

```
public class SymbolsTable {
    private final HashMap<String, ArrayList<SymbolRow>> symbols;
```

As mentioned before we need to handle the scope of the symbols which is why each SymbolRow has a pointer to its parent scope apart from its id, variables can be part of many scopes, not just of functions which also have a name and are also a symbol but of ifs, for loops, while loops, etc. In order to be able to keep track of this we have made these also part of the symbol table, but since we only need the parent scope and id as mentioned previously and we do not want to make them take up unnecessary space in the memory we have made SymbolRow only have those two attributes

```
public class SymbolRow {

    protected SymbolRow parentScope;
    protected int id;
```

In order to then store the actual symbols, we want to retrieve information from we made an actual class called SymbolAttribute that stores all of the necessary attributes and extends SymbolRow, this way they can be stored in the same hash map without a problem.

```
public class SymbolAttribute extends SymbolRow{
    protected final String name;
    protected final String dataType;
    protected ArrayList<Integer> dimensions;
    protected final int declarationLine;
    protected boolean declared;
    protected boolean isFunction;
    protected boolean initialized;
    protected boolean mightNotBeInitialized;
    protected String register;
    protected boolean isReadOnly;
```

The problem with this is that the lookup returns a SymbolRow, not a SymbolAttribute and since we want to abstract this logic from the user (which in this case would be the different modules), instead of putting the getters and setters for the symbol attribute in the SymbolAttribute class, we put them inside the SymbolRow class, so if you perform a getName() on a SymbolRow object, it will cast itself to a SymbolAttribute object and return its name. This allows for the user to modules to not have to implement any additional logic for the way the symbol table is implemented.

```
public String getName() {
    SymbolAttribute symbolAttribute = (SymbolAttribute) this;
    return symbolAttribute.name;
}
```

It should also be taken into account that SymbolRows which are not SymbolAttributes do not have a name but the symbol table requires that the key be a String, therefore the key for this case will be the id converted to a string. Therefore, there are two lookup methods, one that only receives an id, which is used for the previous case mentioned and one that receives an id and name, used for SymbolAttributes.

Parse Tree

The design of the parse tree used for analyzing the input source code is based on the connection of several objects of type *Node*. These were defined with the following attributes:

```
public class Node {
    private final String statement;
    private String value;
    private Node parent;
    private final LinkedList<Node> childs;
    private int id;
    private SourceLocation location;
```

As you can see in the picture above, each Node has a *statement* that specifies the grammar production it represents (either terminal or non-terminal). When dealing with terminal nodes, the *value* attribute holds the input token it references. In the case of non-terminals, the *value* and *statement* fields store the same information. For instance, for the production <main_> both the *statement* and *value* store “main_”, while in the case of a *name* production, a possible *Node* object could have “name” as its statement and “var1” as its *value*. We can also have the opposite scenario when a terminal stores the same *statement* and *value*, which would be the case of a reserved keyword or symbol (“!”, “?”, and “.”, among others).

Additionally, each *Node* has a reference to a *parent Node* object and a list of children *Node*s to portray the hierarchical relationships and establish connections between all the tree entries. This structure allows us to have multiple children to resemble a B-tree structure.

We defined two more attributes that helped us better identify each *Node*: the *location* (declaration coordinates in source file used for error reporting) and an *id*. The latter was used for distinguishing between nodes with the same *value* stored in the Symbols Table (as explained in the module above).

TAC Entry

The intermediate code generator module of our compiler used what we defined as a *TacEntry* class to translate each instruction of the source code from high-level language to an intermediate representation (IR) that could be later used by the target code generator. We choose to express these entries following a TAC (three-address code) representation, as it provides an adequate level of abstraction whilst presenting a simple and consistent structure. The ease of translation from this format to MIPS also proved to be beneficial.

```
private final String arg1;
private final String arg2;
private final String res;
private final Operation op;
private final String tag;
```

Our *TacEntry* structure is formed by five attributes: *arg1, arg2, res, op,* and *tag*. The first two represent the registers that are being used for the operation, specified in the *op* field which is of type Operation (an enumeration with all the possible allowed operations). The result of this calculation will be stored in the register specified in the *res* attribute. Lastly, we make use of the *tag* field for specific scenarios in which we can benefit from a label, for example: when declaring a variable or function, doing function calls, or defining parameters of a function.

It is worth mentioning that the use of all the attributes is not required for defining an operation. We evaluate how a specific instruction will be implemented in a TAC format and decide how to express it. For instance, a function declaration would only use *arg1* for specifying the return datatype of the function, including the name of the function in the *tag* field, having *FUNC_DEC* as its *op*. Another example could be the equality of two registers, which would be expressed as an *EQUAL* operation stored in *op*, including the source and destination registers in *arg1* and *res* attributes respectively.

Error Handler

This component implements the *ErrorListener* class and is the one responsible for storing the errors and warning encountered during compilation. This one has four *StringBuilder* attributes that hold these respective errors.

```
public class ErrorHandler implements ErrorListener {

    private final StringBuilder lexicalErrors;
    private final StringBuilder syntaxErrors;
    private final StringBuilder semanticErrors;
    private final StringBuilder semanticWarnings;
```

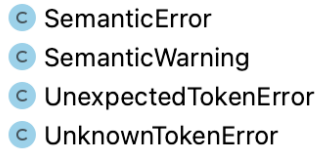
The set of errors are stored in separate attributes as each of them has their own representation. Whenever an error is encountered, we make use of the *report* method, which receives an error as a parameter and adds it to its corresponding list (with its personalized header). You can see the *Error* class created for this purpose in the picture below. This one defines the four possible error types (including warnings) and the *SourceLocation* of the error (number of column and line declaration).

```
public abstract class Error extends Throwable {

    Miguel-Nicolás Palau Lessa +1
    public enum ErrorType {
        LEXICAL,
        SYNTAX,
        SEMANTIC,
        WARNING
    }

    private final ErrorType type;
    private final SourceLocation location;
```

To properly define each type of error with its corresponding attributes, for classes were created extending this one and overriding the *getMessage* method. Below you can see the example of the *UnexpectedTokenError* which defines the error message, and the attributes of received and expected tokens to provide the user with as much useful information as possible for them to fix their mistake.




```
public class UnexpectedTokenError extends Error {

    private static final String MESSAGE = "Unexpected token '%s', expected '%s'";
    private final Token token;
    private final String expectedType;
```

Once the program wants to be terminated because an error without possibility of recovery has been found, or because the compilation process is done, we call the following method.

```
public String getErrorWall() {
    String errors = lexicalErrors.toString() + syntaxErrors + semanticErrors + semanticWarnings;
    if (errors.isEmpty()) {
        return "Stan 🎉🎉🎉";
    } else {
        return errors;
    }
}
```

This one is in charge of concatenating all the encountered errors and returning them as a whole or returning the message "*Stan*" whenever no errors or warning were found. The returned value of this function is then printed into the terminal so that the user has useful insights about the mistakes found in their code.

Below you can see some of our personalized headers for the different types of errors:

```
if (semanticErrors.isEmpty()) {
    semanticErrors.append("------------------------------ What a boomer ¬_¬ ------------------------------\n\n");

if (semanticWarnings.isEmpty()) {
    semanticWarnings.append("------------------------------ A little sus ◔_◔ ------------------------------\n\n");
}

if (lexicalErrors.isEmpty()) {
    lexicalErrors.append("------------------------------ Too cringe ✖_✖------------------------------\n\n");
}
if (syntaxErrors.isEmpty()) {
    syntaxErrors.append("------------------------------ Ohh, you're canceled :) ------------------------------\n\n");
}
```

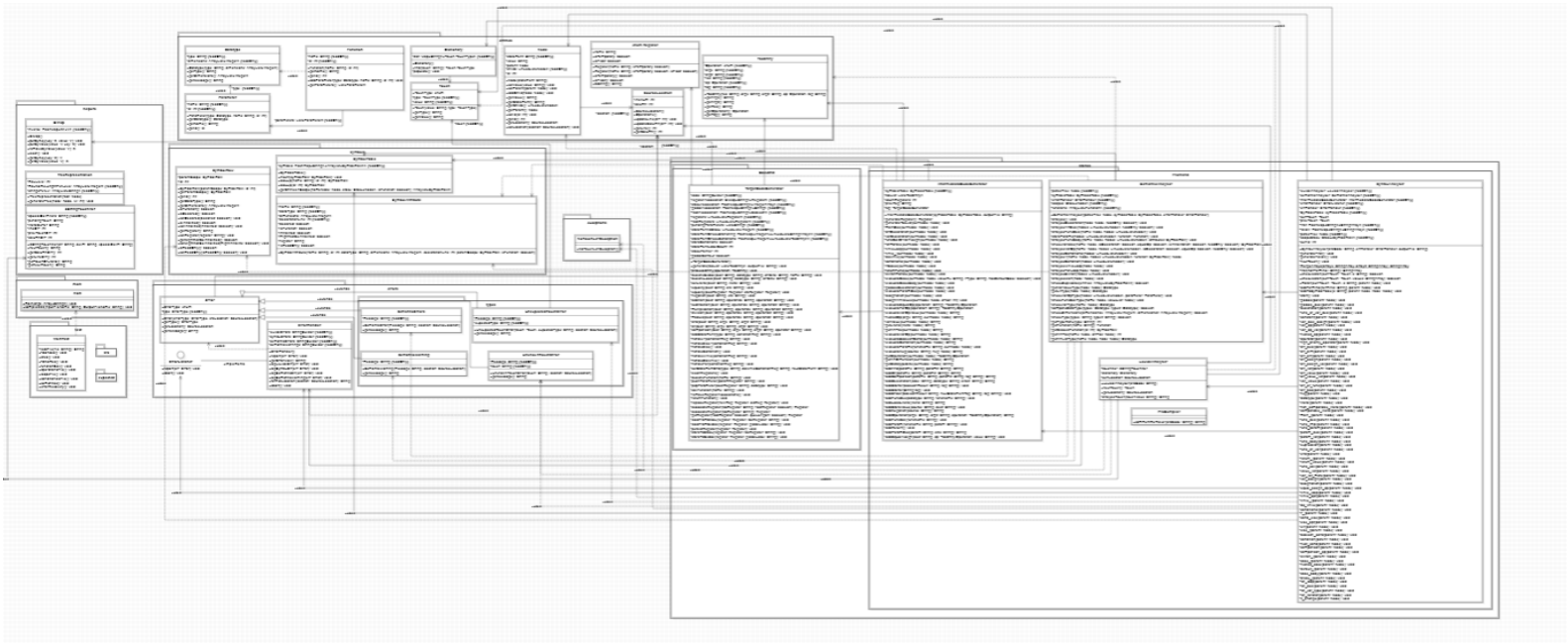
Conclusion

Now that we have successfully created our own programming language (lol), we can truly state that this achievement marks a significant milestone in our academic lives. Throughout this project, we not only applied concepts learned in class, but we also had full control of the design, allowing us to express our creativity. Despite experiencing ups and downs, our team's commitment was never the issue, our passion for bringing this project to fruition is evident in every aspect of the work.

The journey was at times long and tedious, however from the conceptualization to the final stages of development, it rigorously tested our skills and pushed our boundaries. Although the backend was unable to handle arrays, we are immensely proud of the expansion in our capabilities that this project has brought on us.

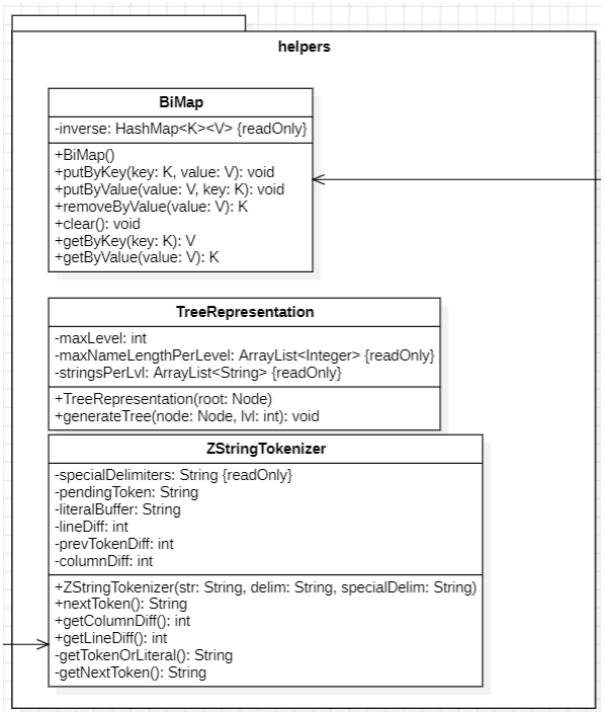
Let’s continue to build a better future together, one line of code at a time!

Class Diagram

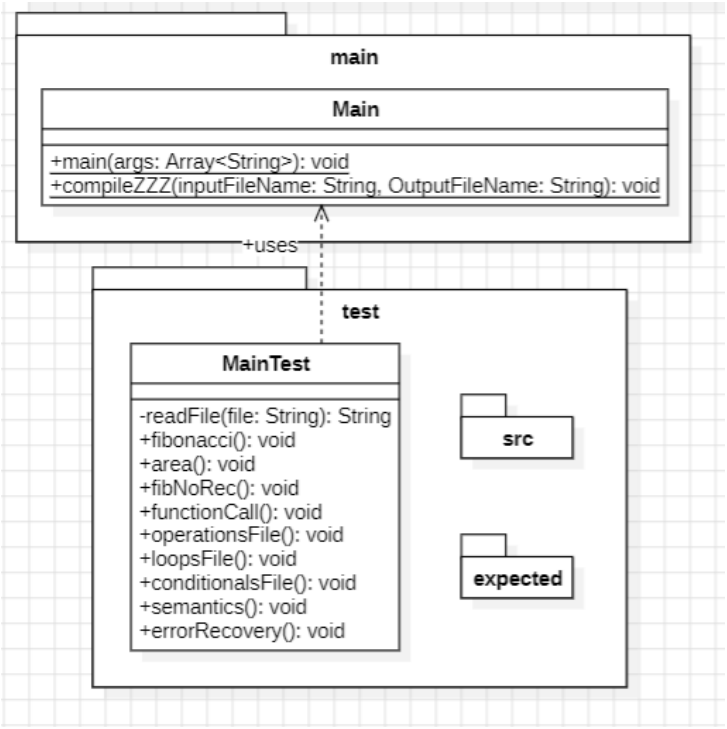


The code is organized in different packages: Helpers, Test, Main, Exceptions, Errors, Symbols, Entities and Stages.

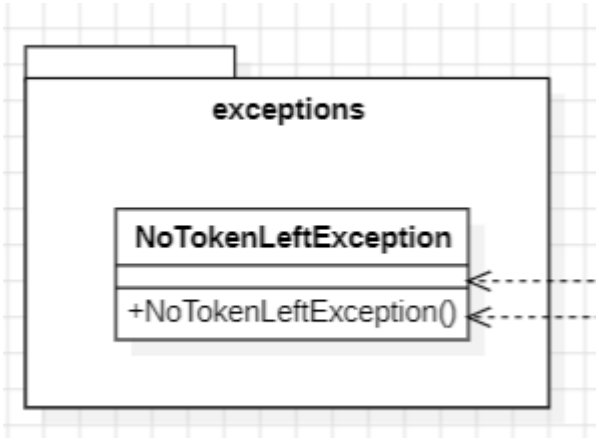
The Helpers directory stores different classes that implement different functionalities used to ease the development of different stages:



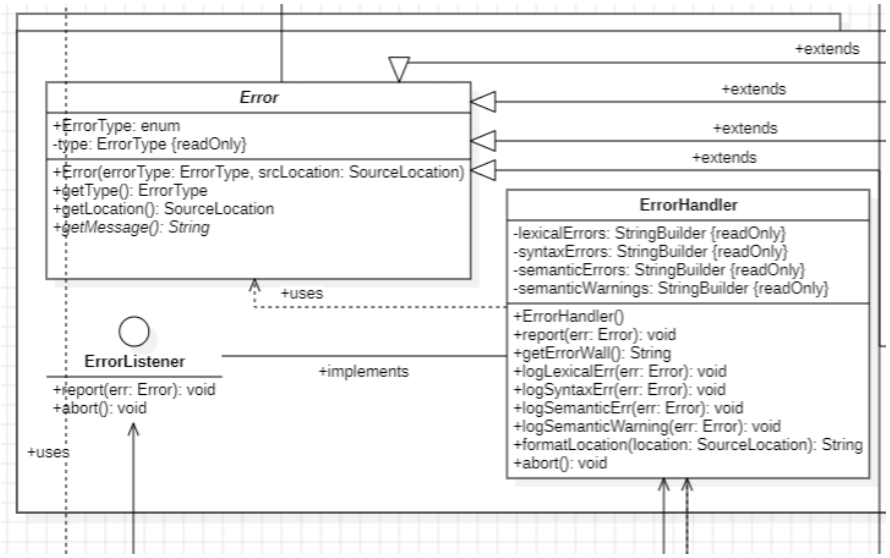
The Main directory stores the Main class used to run the project and the Test directory stores the class and files used to test that the compiler works correctly either for compiling and detecting errors:

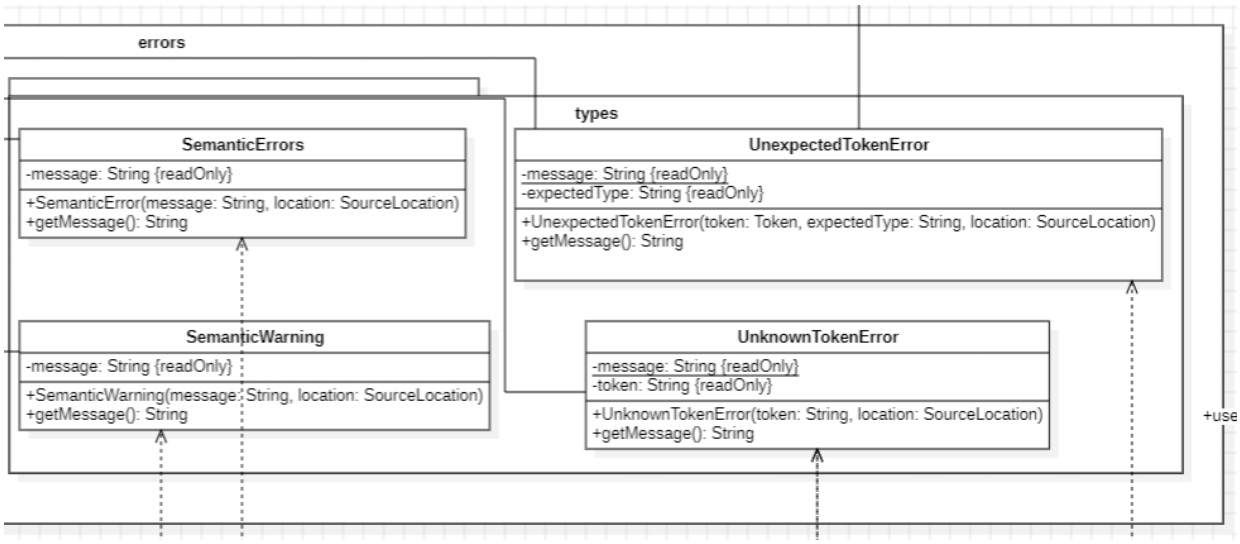


The Exceptions directory stores the exceptions thrown:

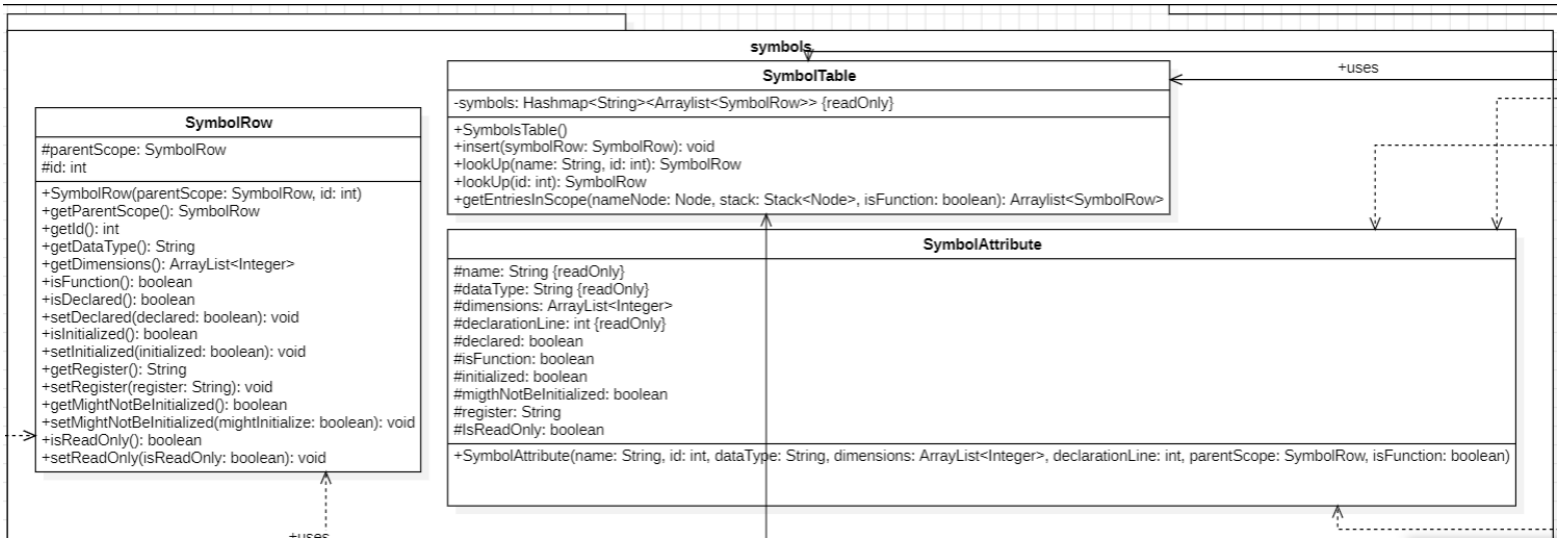


The Errors directory stores the class used to handle errors and the different types of errors that can be detected:

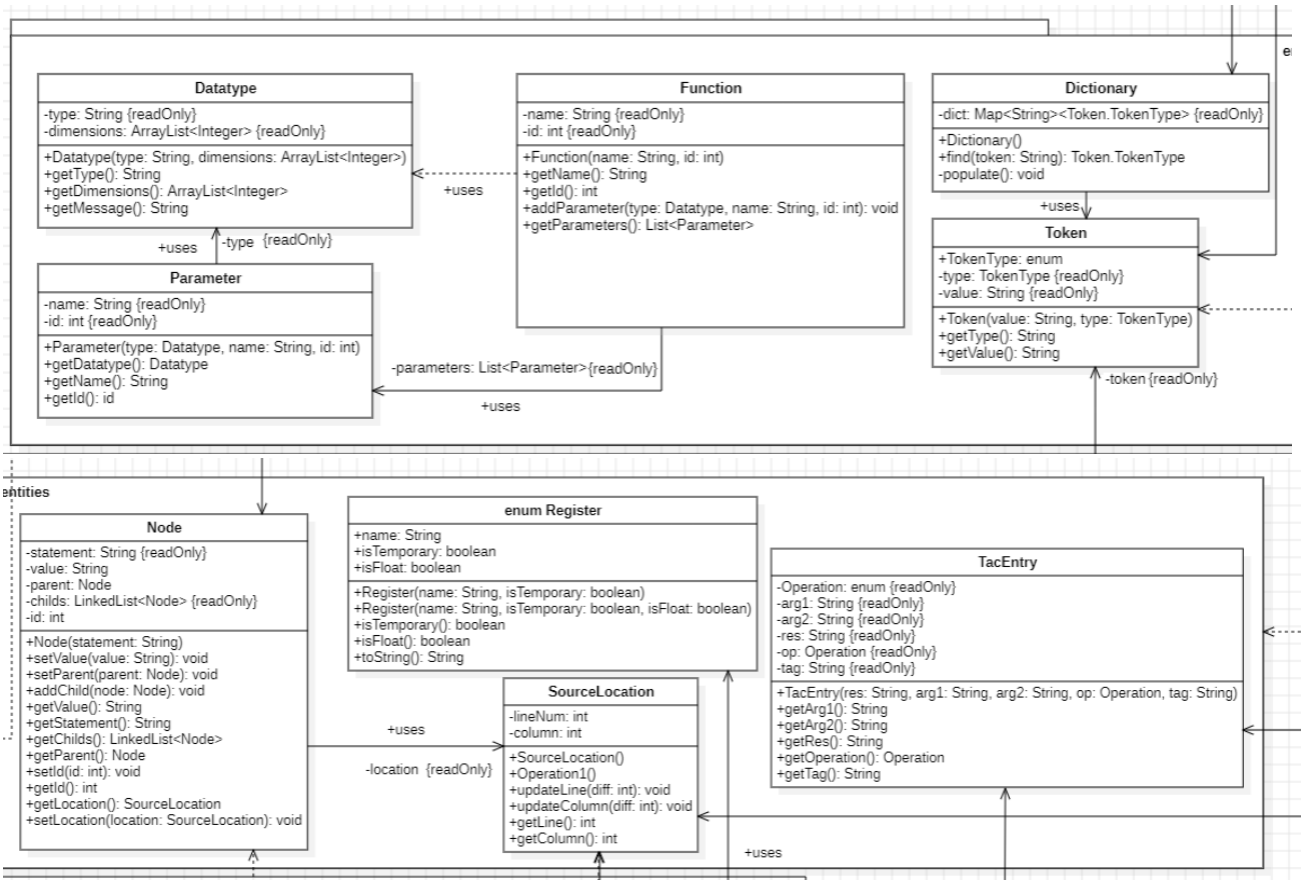




The Symbols directory stores the class implementing the symbols table as well as two other classes used to create objects stored in the symbols table:

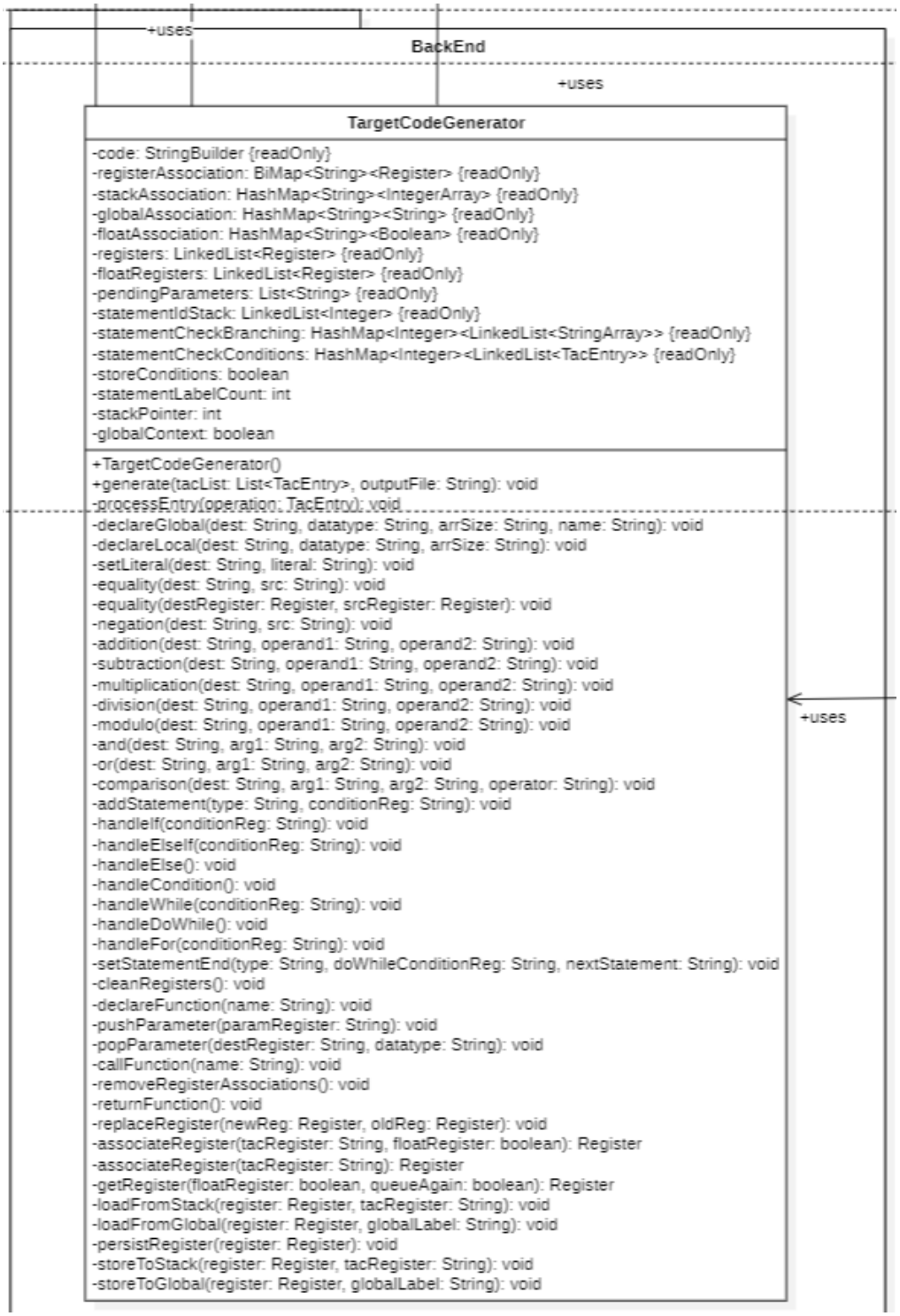


The Entities directory stores the different classes used to generate different objects needed throughout the whole project:

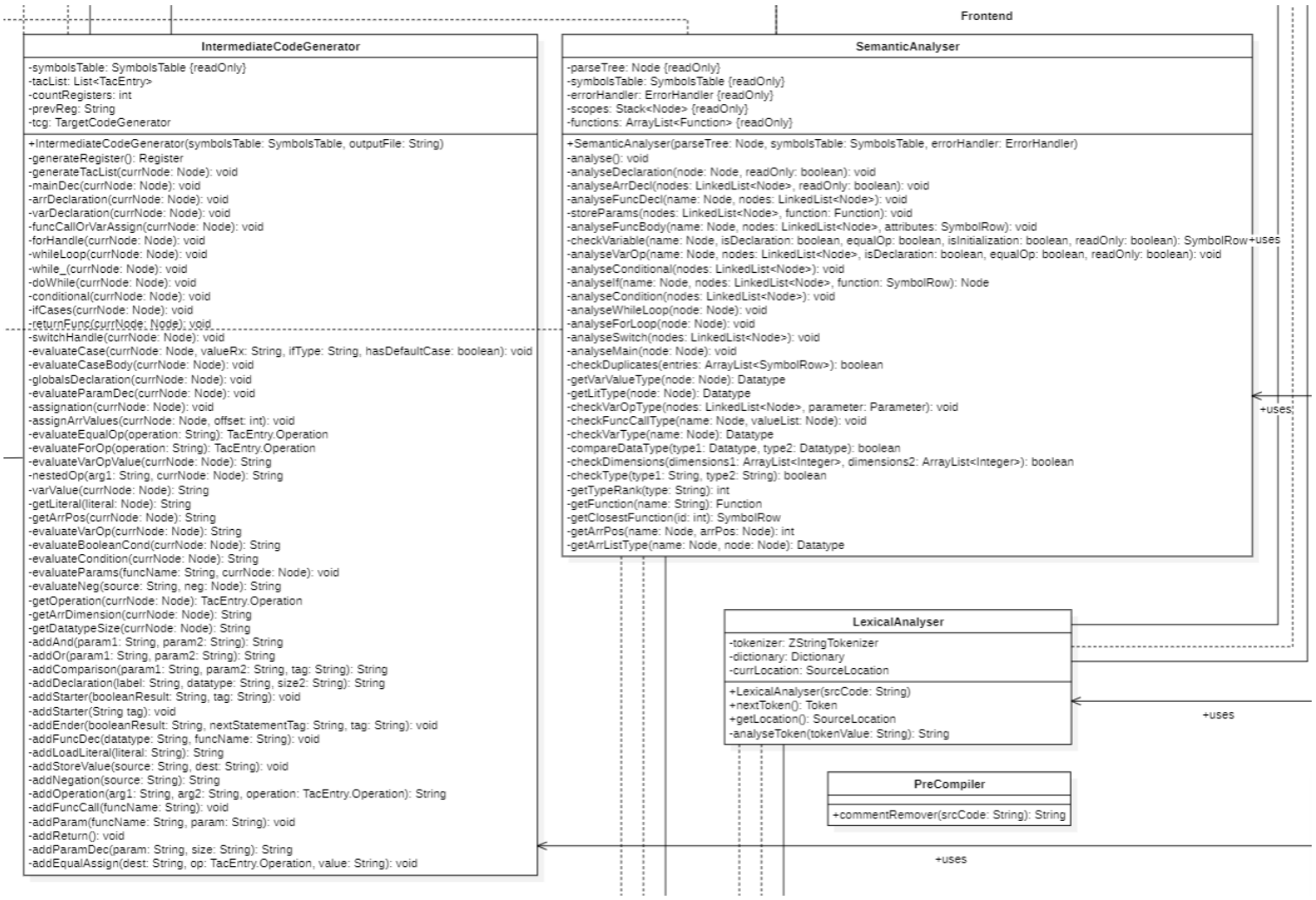


And, finally, Stages directory stores the class implementing the functionalities needed before compilation and two other directories, the backend and the frontend.

The Backend directory which stores the TargetCodeGenerator class, used to generate the executable:



The frontend directory, which stores the lexical, syntax and semantical analyzers as well as the intermediate code generator:



SyntaxAnalyser

-lexicalAnalyser: LexicalAnalyser {readOnly}
-semanticAnalyser: SemanticAnalyser {readOnly}
-intermediateCodeGenerator: IntermediateCodeGenerator {readOnly}
-errorHandler: ErrorListener {readOnly}
-errHandler: ErrorHandler {readOnly}
-SymbolsTable: symbolsTable {readOnly}
-currToken: Token
-prevToken: Token
-first: HashMap<String><StringArray> {readOnly}
-follow: HashMap<String><StringArray> {readOnly}
-parseTree: Node {readOnly}
-scopeStack: Stack<SymbolRow> {readOnly}
-currId: int

+SyntaxAnalyser(srcCode: String, errHandler: ErrorHandler, outputFile: String)
-generateFirst(): void
-generateFollow(): void
-nextToken(): void
-mergeArrays(array1: StringArray, array2: StringArray): StringArray
-first(nonTerminal: String): StringArray
+checkMatch(currToken: Token, s: String): boolean
+checkMatch(currToken: Token, values: StringArray): boolean
+match(currToken: Token, s: String, parent: Node): void
-addTerminal(terminal: String, parent: Node): void
-addToSymbolTable(s: String, parent: Node, node: Node): void
-start(): void
-globals(parent: Node): void
-globals_decl(parent: Node): void
-declaration(parent: Node): void
-func_or_var_decl(parent: Node): void
-constant(parent: Node): void
-var_decl_assign(parent: Node): void
-var_op(parent: Node): void
-var_op_val(parent: Node): void
-nested_op(parent: Node): void
-operator(parent: Node): void
-high_priority_operator(parent: Node): void
-arr_decl(parent: Node): void
-arr_dim(parent: Node): void
-arr_arr(parent: Node): void
-arr_assign(parent: Node): void
-arr_assign_val(parent: Node): void
-arr_list(parent: Node): void
-arr_value(parent: Node): void
-arr_value_list(parent: Node): void
-var_value(parent: Node): void
-arr_or_func(parent: Node): void
-arr_pos(parent: Node): void
-neg(parent: Node): void
-datatype(parent: Node): void
-literal(parent: Node): void
-non_comparable_literal(parent: Node): void
-comparable_literal(parent: Node): void
-main_(parent: Node): void
-func_decl(parent: Node): void
-func_impl(parent: Node): void
-func_param(parent: Node): void
-param_decl(parent: Node): void
-param_list(parent: Node): void
-func_body(parent: Node): void
-expression(parent: Node): void
-func_or_var(parent: Node): void
-end(parent: Node): void
-return_(parent: Node): void
-return_value(parent: Node): void
-func_call(parent: Node): void
-value_list(parent: Node): void
-var_list_more(parent: Node): void
-var_assign(parent: Node): void
-assignation(parent: Node): void
-equal_assign_op(parent: Node): void
-while_loop(parent: Node): void
-while_opt(parent: Node): void
-while_(parent: Node): void
-do_while(parent: Node): void
-conditional(parent: Node): void
-if_(parent: Node): void
-cond_else(parent: Node): void
-else_opt(parent: Node): void
-elif(parent: Node): void
-else_(parent: Node): void
-boolean_cond(parent: Node): void
-condition(parent: Node): void
-nest_cond(parent: Node): void
-comparison(parent: Node): void
-comparison_op(parent: Node): void
-switch_(parent: Node): void
-case_(parent: Node): void
-nested_case(parent: Node): void
-default_(parent: Node): void
-case_body(parent: Node): void
-break_(parent: Node): void
-for_loop(parent: Node): void
-for_decl(parent: Node): void
-for_var_type(parent: Node): void
-for_iterator(parent: Node): void
-it_change(parent: Node): void

Examples

Below you will find a set of example codes written in Z language that our compiler is able to successfully process and run.

Fibonacci

The following program computes the Fibonacci sequence of the number declared in the CEO function by recursively calling the `Fibonacci` function.

```
<< Program to compute fibonacci sequence
    of a number
   Author: Marc
   Version: 3.33
   Date: 7-5-2023 >>

<< Globals declaration >>
fact bro ZERO = 0.
fact bro ONE = 1.

<< Fibonacci sequence >>
bro fibonacci ¿ bro num ? ¡
   swipe ¿ num ? ¡
       right 0:
         throwback ZERO.
      right 1:
          throwback ONE.
      left:
          bro fVal = fibonacci ¿num - 1?.
          fVal += fibonacci ¿num - 2?.
         throwback fVal.
   !
   throwback ZERO.
 !

<< Main function of the program
   Author: Adrian
   Version: 3.33
   Date: 2-3-2023 >>
CEO ¡
    bro num = 6.
    << Compute fibonacci >>
   bro fib = fibonacci¿ num ?.
!
```

Fibonacci no recursion

This example code is the equivalent of the previous one without using recursion, showing the use conditionals and loops.

```
<< Program to compute fibonacci sequence
    of a number
   Author: Marc
   Version: 3.33
   Date: 7-5-2023 >>

bro fibonacci ¿bro num? ¡
    bro fib0 = 0.
    bro fib1 = 1.
    bro nextFib = 0.

    like ¿ num == 0? ¡
        nextFib = 0.
    !
    like ¿ num == 1? ¡
        nextFib = 1.
    !

    4 ¿bro i = 2. i <= num. i++? ¡
        nextFib = fib0 + fib1.

        fib0 = fib1.
        fib1 = nextFib.
    !
    throwback nextFib.
!

<< Main function >>
CEO ¡
    bro res = fibonacci ¿3?.
!
```

Area

The following program computes the area of the figure (‘S’ for square, ‘T’ for triangle and ‘C’ for circle) specified in the first parameter of the `computeArea` function using the other parameters.

```
<< Program to compute the area of a given figure
   Author: Andrea
   Version: 3.33
   Date: 15-05-2024 >>

<< Globals declaration >>
sis area = 0.
fact sis PI = 3´14.

<< Area computation >>
zombie computeArea ¿ mainchar shape, sis width, sis height, sis radius ? ¡
    swipe ¿ shape ? ¡
        right 'S': <<Square>>
            area = height * 2.
            periodt.

        right 'T': <<Triangle>>
            area = height * width / 2.
            periodt.

        right 'C': <<Circle>>
            area = PI * radius * radius.
            periodt.
    !
    throwback ghosted.
!

<< Main function >>
CEO ¡
    << Define measures >>
    sis width = 12.
    sis height = 7.
    sis radius = 4´3.

    << Determine area >>
    computeArea ¿ 'S', width, height, radius ?.
    sis res = area.
!
```

Average grade

This code shows how the use of arrays would be in Z language. However, remember that these were not fully implemented in the last section of our compiler (Target Code Generator), so this program is unable to run. However, we believe it is relevant to show the syntax and see the overall use of Z.

```
<< Program to compute happiness level according to average grades>>
bro MAX = 100.
bro MEH = 50.
bro NONE = 0.

<< Passing calculation >>
bipolar goodEnough ¿ sis average ? ¡
    like ¿ average >= 5 ? ¡
        throwback smash.
    !
    throwback pass.
!

<< Principal function of the program
   Author: Adrian
   Version: 3.33
   Date: 2-3-2023 >>
CEO ¡
   fam 10 sis grades = [ 2´9, 5´8, 8´0, 7´1, 5´2, 4´9, 6´9, 5´7, 6´3, 7´5 ] .

   << Average grade calculation >>
   sis average = 0 .
   4 ¿ bro i = 0 . i < 10 . i++ ? ¡
      average += grades[i] .
   !
   average /= 10 .

   << Check the happiness level of the student >>
    sis happiness.
    bipolar passed = goodEnough ¿average?.

   like ¿ passed == smash & average >= 9 ? ¡
      happiness = MAX.
   ! whatever like ¿ passed == smash ? ¡
       happiness = MEH.
   ! whatever ¡
      happiness = NONE.
   !
!
```

Bibliography

A. Patterson, D., & L. Hennessy, J. (n.d.). *Computer Organization and Design* (4th ed.) [PDF]. Morgan Kauffman. https://nsec.sjtu.edu.cn/data/MK.Computer.Organization.and.Design.4th.Edition.Oct.2011.pdf

University of California, Santa Barbara. (n.d.). Lecture 8: Procedure + Nested Procedure. [PDF]. UCSB Computer Science Department. https://sites.cs.ucsb.edu/~htzheng/teach/cs64s11/pdf/lecture8.pdf

Çelikkanat, H. (2007). Introduction to MIPS and SPIM. [PDF]. METU Department of Computer Engineering. https://saksagan.ceng.metu.edu.tr/courses/ceng444/link/444_phase2_recitation.pdf

Pietro. (2019, 13 October). Implementation of Semantic Analysis. Compilers. https://pgrandinetti.github.io/compilers/page/implementation-semantic-analysis/

Sewell, P. (2014-2015). Semantics of Programming Languages. [PDF]. Computer Science Tripos, Part 1B. University of Cambridge. https://www.cl.cam.ac.uk/teaching/1415/Semantics/notes.pdf