The WACC Language Specification

Second Year Computing Laboratory
Department of Computing
Imperial College London

1 What is WACC?

WACC (pronounced "whack") is a simple variant on the While family of languages encountered in many program reasoning/verification courses (in particular in the Models of Computation course taught to our 2nd year undergraduates). It features all of the common language constructs you would expect of a While-like language, such as program variables, simple expressions, conditional branching, looping and no-ops. It also features a rich set of extra constructs, such as simple types, functions, arrays and basic tuple creation on the heap.

The WACC language is intended to help unify the material taught in our more theoretical courses (such as Models of Computation) with the material taught in our more practical courses (such as Compilers). The core of the language should be simple enough to reason about and the extensions should pose some interesting challenges and design choices for anyone implementing it.

2 WACC Language Syntax

We give the syntax of the WACC language in Backus-Naur Form (BNF) extended with some basic regular expression notation that simplifies the presentation:

- (x-y) stands for 'range', meaning any value from x to y inclusive;
- (x)? stands for 'optional', meaning that x can occur zero or one times;
- (x)+ stands for 'repeatable', meaning that x can occur one or more times;
- (x)* stands for 'optional and repeatable', meaning that x can occur zero or more times.

2.1 BNF

```
⟨program⟩       ::= 'begin' ⟨func⟩* ⟨stat⟩ 'end'

⟨func⟩          ::= ⟨type⟩ ⟨ident⟩ '(' ⟨param-list⟩? ')' 'is' ⟨stat⟩ 'end'

⟨param-list⟩    ::= ⟨param⟩ ( ',' ⟨param⟩ )*

⟨param⟩         ::= ⟨type⟩ ⟨ident⟩

⟨stat⟩          ::= 'skip'
                  | ⟨type⟩ ⟨ident⟩ '=' ⟨assign-rhs⟩
                  | ⟨assign-lhs⟩ '=' ⟨assign-rhs⟩
                  | 'read' ⟨assign-lhs⟩
                  | 'free' ⟨expr⟩
                  | 'return' ⟨expr⟩
                  | 'exit' ⟨expr⟩
                  | 'print' ⟨expr⟩
                  | 'println' ⟨expr⟩
```

```
                     | 'if' ⟨expr⟩ 'then' ⟨stat⟩ 'else' ⟨stat⟩ 'fi'
                     | 'while' ⟨expr⟩ 'do' ⟨stat⟩ 'done'
                     | 'begin' ⟨stat⟩ 'end'
                     | ⟨stat⟩ ';' ⟨stat⟩

⟨assign-lhs⟩        ::= ⟨ident⟩
                     | ⟨array-elem⟩
                     | ⟨pair-elem⟩

⟨assign-rhs⟩        ::= ⟨expr⟩
                     | ⟨array-liter⟩
                     | 'newpair' '(' ⟨expr⟩ ',' ⟨expr⟩ ')'
                     | ⟨pair-elem⟩
                     | 'call' ⟨ident⟩ '(' ⟨arg-list⟩? ')'

⟨arg-list⟩          ::= ⟨expr⟩ (',' ⟨expr⟩ )*

⟨pair-elem⟩         ::= 'fst' ⟨expr⟩
                     | 'snd' ⟨expr⟩

⟨type⟩              ::= ⟨base-type⟩
                     | ⟨array-type⟩
                     | ⟨pair-type⟩

⟨base-type⟩         ::= 'int'
                     | 'bool'
                     | 'char'
                     | 'string'

⟨array-type⟩        ::= ⟨type⟩ '[' ']'

⟨pair-type⟩         ::= 'pair' '(' ⟨pair-elem-type⟩ ',' ⟨pair-elem-type⟩ ')'

⟨pair-elem-type⟩    ::= ⟨base-type⟩
                     | ⟨array-type⟩
                     | 'pair'

⟨expr⟩              ::= ⟨int-liter⟩
                     | ⟨bool-liter⟩
                     | ⟨char-liter⟩
                     | ⟨str-liter⟩
                     | ⟨pair-liter⟩
                     | ⟨ident⟩
                     | ⟨array-elem⟩
                     | ⟨unary-oper⟩ ⟨expr⟩
                     | ⟨expr⟩ ⟨binary-oper⟩ ⟨expr⟩
                     | '(' ⟨expr⟩ ')'

⟨unary-oper⟩        ::= '!' | '-' | 'len' | 'ord' | 'chr'

⟨binary-oper⟩       ::= '*' | '/' | '%' | '+' | '-' | '>' | '>=' | '<' | '<=' | '==' | '!=' | '&&' | '||'

⟨ident⟩             ::= ( '_' | 'a'-'z' | 'A'-'Z' ) ( '_' | 'a'-'z' | 'A'-'Z' | '0'-'9' )*

⟨array-elem⟩        ::= ⟨ident⟩ ('[' ⟨expr⟩ ']')+
```

⟨*int-liter*⟩	::=	⟨*int-sign*⟩? ⟨*digit*⟩+
⟨*digit*⟩	::=	(‘`0`’-‘`9`’)
⟨*int-sign*⟩	::=	‘`+`’ | ‘`-`’
⟨*bool-liter*⟩	::=	‘`true`’ | ‘`false`’
⟨*char-liter*⟩	::=	‘`'`’ ⟨*character*⟩ ‘`'`’
⟨*str-liter*⟩	::=	‘`"`’ ⟨*character*⟩* ‘`"`’
⟨*character*⟩	::=	`any-ASCII-character-except-‘\’-‘'’-‘"’`
	|	‘`\`’ ⟨*escaped-char*⟩
⟨*escaped-char*⟩	::=	‘`0`’ | ‘`b`’ | ‘`t`’ | ‘`n`’ | ‘`f`’ | ‘`r`’ | ‘`"`’ | ‘`'`’ | ‘`\`’
⟨*array-liter*⟩	::=	‘`[`’ (⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)*)? ‘`]`’
⟨*pair-liter*⟩	::=	‘`null`’
⟨*comment*⟩	::=	‘`#`’ (`any-Extension:{ASCII-}character-except-EOL`)* ⟨*EOL*⟩

NB: There is an additional constraint on the syntax of function definitions, that every execution path through the body of the function must end with either a `return` statement or an `exit` statement.
Extension:**NB:** Comments can go anywhere in the document except for inside a ⟨*str-liter*⟩.

3 WACC Language Semantics

We now go through each of the language components and explain their behaviour and purpose in more detail.

3.1 Types

With the exception of nested pairs, the WACC language is both statically and strongly typed.

- Static, in that, once declared, the type of a variable is fixed for the duration of the program.
- Strong, in that the compiler should not coerce between types.

There is also no explicit typecasting.

Basic Types: The basic types in the WACC language are:

- `int`: The Integer type. Integers in the WACC language can take any value from -2^{31} to $2^{31} - 1$ inclusive.
- `bool`: The Boolean type (`true` or `false`).
- `char`: The Character type. The WACC language supports only the ASCII characters.
- `string`: The String type.

We write `T` to denote an arbitrary type.

Arrays: As well as the basic types given above, the WACC language also supports the array type. We write `T[]` to denote an array whose elements are of type `T`. Note that `T` can be of any type, including another array type, which allows for nested arrays. In the WACC language, arrays of characters are treated like strings. Arrays are allocated on the heap. As well as their elements, each array also tracks its length, which is set when it is created.

Pairs: Pairs are allocated on the heap and contain two elements that can be of any type. We write `pair(`$\mathtt{T}_1$`, `$\mathtt{T}_2$`)` to donate a pair whose first element is of type $\mathtt{T}_1$ and second element is of type $\mathtt{T}_2$ (these need not be the same). Note that if either $\mathtt{T}_1$ or $\mathtt{T}_2$ is a pair type, we do not write the type of the sub-elements. For example, a pair whose first element is an integer and whose second element is a pair of characters is written as `pair(int, pair)` and not as `pair(int, pair(char, char))`. It is obvious that we lose some typing information in this way. Moreover, due to the loss of type information for nested pairs, it is possible to subtly coerce between types.

3.2 Program Scopes

The WACC language includes explicit scoping. Various statements introduce new program scopes, which have an effect on the visibility of program variables.

Whenever a new variable is declared it is added to the current program scope. When a program scope is exited, every variable created within that scope is destroyed. This means that variables are not accessible by statements outside the scope of their creation, although they are accessible in child scopes.

The main, or global scope is created at the start of a WACC program and is exited at the end of the program. Functions can only be created at the beginning of this global scope, but they may be called from within any child scope.

We will see that several other program constructs, including functions, while loops and conditional branches, introduce new program scopes during their execution.

3.3 Programs

A WACC program ⟨*program*⟩ consists of zero or more function definitions followed by the body of the main function. The whole program is written between the `begin` and `end` tokens, denoting the main or global program scope. A WACC file (extension `.wacc`) only ever contains a single WACC program.

3.4 Function Definitions

A function definition ⟨*func*⟩ consists of a return type, a function name and zero or more typed parameters followed by the function's body. A function's body, which is denoted by the `is` and `end` tokens, is executed in its own scope containing only the parameters passed into the function. As mentioned above, any execution path through the body of the function must end with either a `return` statement, whose expression type must match the function's return type, or an `exit` statement.

Functions can only be defined at the beginning of the global scope, before the body of the main function. Functions may, however, be both recursive and mutually recursive.

3.5 Statements

A statement ⟨*stat*⟩ consists of: a no-op, a variable definition, an assignment, an input read, a memory free, a function return, an exit call, a print command, a conditional branch, a while loop, a scope introduction or the sequential composition of two statements.

We discuss each of these in more detail below.

No-op Statements: A no-op statement `skip` does not do anything. It is used where a statement is expected but we do not want to do anything. For example, in an `if` statement where we want to have an empty `else` clause.

Variable Declaration Statements: A variable declaration statement creates a new program variable in the current scope setting its static type and initial value. The statement must be given a valid WACC type ⟨*type*⟩, a variable name ⟨*ident*⟩ and an initial assignment value ⟨*assign-rhs*⟩.

Variable names must not clash with any other identifiers within the scope, including previously declared function or variable names and keywords.

They can consist of underscores, upper or lowercase letters and digits, however they cannot start with a digit and must have at least one character.

The initial assignment to a variable follows all of the assignment restrictions discussed in detail in the assignment statement section below.

A variable must be declared before it is referenced in an expression or assignment.

Any attempt to access an undeclared variable results in a semantic compiler error stating which variable is not defined, and giving an exit code of 200.

Additionally, every use of a variable must match the type assigned to that variable when it was declared.

A variable can only be accessed within the scope of its declaration (or any child scope) and it is destroyed when exiting this scope. Variables must be unique within their scope, so a variable cannot be redefined within the same scope. Variables can, however, be redefined within a child scope. In this case any access to the variable will reference the newest scope's definition.

Once the child scope is exited the variable returns to its previous definition before it was redefined in the child scope.

Gap 1. *Complete the above paragraph* (**5 marks**)

Assignment Statements: An assignment statement updates its target (the left-hand side of the `=`) with a new value (the right-hand side of the `=`). The target of an assignment can be either a program variable, an array element or a pair element. The assignment value can be one of five possible types: an expression, an array literal, a function call, a pair constructor or a pair element.

- If the assignment value is an expression ⟨*expr*⟩ then the target and the expression must have the same type. The expression is then evaluated and the resulting value is copied into the target.

- If the assignment value is an array literal ⟨*array-liter*⟩ then the target and value arrays must have the same element type, but the length can be different. The value array is then allocated on the heap with each element initialised to the given value. After that, the reference of the value array is copied to the reference of the target array. For more details on array literals, see the expressions section.

- If the assignment value is a function call `call` then the target and function return value must have the same type. The number and types of the function's arguments must also match the function's definition. A function is called by its name and its arguments are passed by value (for basic types) or by reference (for arrays and pairs). When called, the function's body is executed in a new scope, not related to the current scope. The only declared variables are the function's parameters, whose types are set by the function definition and whose values are set by the function call's arguments. When the execution of the function body terminates, the function's return value is then copied into the assignment target.

- If the assignment value is pair constructor `newpair` then the target must be of type `pair(`T_1`,` T_2`)`. The pair constructor is passed two expressions that must match T_1 and T_2 respectively. A `newpair` assignment allocates enough memory on the heap to store the pair structure and its elements. It then initialises each element of the pair using the evaluation of the first expression for the first element and the evaluation of the second expression for the second element. Pairs, in the WACC language, are always used by reference, so a reference to the pair is copied into the target, rather than the actual content of the pair.

- If the assignment value is a pair element ⟨*pair-elem*⟩ then the expression passed to the pair element must be of type `pair` and the target must have the same type as the first (or second) element of the pair when using `fst` (or `snd`) keyword. The pair expression is evaluated to obtain a reference to a pair and this is dereferenced to find the corresponding pair element, which is then copied into the target.

Read Statements: A read statement `read` is a special assignment statement that takes its value from the standard input and writes it to its argument. Unlike a general assignment statement, a read statement can only target a program variable, an array element or a pair element. Additionally, the read statement can only handle character or integer input.

The read statement determines how it will interpret the value from the standard input based on the type of the target. For example, if the target is a variable of type `int` then it will convert the input string into an integer.

Memory Free Statements: A memory free statement `free` is used to free the heap memory allocated for a pair or array and its immediate content. The statement is given an expression that must be of type `pair(`$\mathtt{T}_1$`, `$\mathtt{T}_2$`)` or `T[]` (for some `T`, $\mathtt{T}_1$, $\mathtt{T}_2$). The expression must evaluate to a valid reference to a pair or array, otherwise a segmentation fault will occur at runtime.

If the reference is valid, then the memory for each element of the pair/array is freed, so long as the element is not a reference to another pair or another array (i.e. free is not recursive). Then the memory that stores the pair/array itself is also freed.

Function Return Statements: A return statement can only be present in the body of a non-main function and is used to return a value from that function. The type of the expression given to the return statement must match the return type of the function. Once the return statement is executed, the function is immediately exited.

Exit Statements: An exit statemnt `exit` may be present anywhere and is used to immediately exit a program with a given exit code (any code after the exit statement will not be executed). The type of the expression given to the exit statement must be `int` (any other type will result in a semantic error at compile time). The exit code will be equal to the provided expression, modulo 256.

Gap 2. *define exit statements*	(**2 marks**)

Print Statements: There are two types of print command in the WACC language. The `print` command takes an expression and prints the result of its evaluation to the standard output. The `println` command is similar, but additionally prints out a new line afterwards.

The output representation of each expression evaluation depends on the type of the expression. The behaviour of the print statements for each type of expression is shown in Table **??**, along with some example cases.

Gap 3. *Fill in Table* ??	(**3 marks**)

Conditional Branch Statements: A conditional branch statement `if` evaluates an expression and determines which program path to follow. The statement is given a condition expression, that must be of type `bool`, and two body statements, one for the `then` branch and one for the `else` branch.

If the condition evaluates to `true`, then the `then` body statement is executed. Otherwise, the `else` body statement is executed. Each of the program branches is executed in its own scope, which are denoted by the `then` and `else` tokens and the `else` and `fi` tokens, respectively.

[1]This is not exactly an expression because it can only appears on the right hand side of an assignment. However, it gives the best example here.

Expression Type	Behaviour	Example Expression	Example Output
`int`	Output the integer converted to a decimal string.	`10`	“10”
`bool`	Output “true” if the boolean is `true` and “false” otherwise.	`false`	“false”
`char`	Output a single-character string.	`'c'`	“c”
`string` or `char[]`	Output the string	`"foobar"`	“foobar”
Other Array Types	Output the memory address of the array in hexadecimal format	`[0, 1, 2]`	“0x23010”
`pair`	Ouput the memory address of the pair in hexadecimal formate	`newpair(a, b)`[1]	“0x23010”

Table 1: The behaviour of the print statements for each type of expression.

While Loop Statements: A while loop statement `while` evaluates an expression and determines whether to execute a statement or not. The statement is given a condition expression, that must be of type `bool`, and one body statement (which may be a sequential composition of statements), which is written between a `do` and a `done` token.

If the condition evaluates to `true`, then the program will execute the body statement and will then execute the full while statement another time. Otherwise, if the condition statement evaluates to `false`, nothing is executed and the program continues after the `done` token. The body is executed in its own scope, denoted by the `do` and `done` tokens.

-E.g. Suppose `p` is a valid WACC expression, which evaluates to a `bool`, and `A` is a valid WACC statement. Then `while p do A done` means:

- If `p` evaluates to `true`, then execute `A; while p do A done`.
- If `p` evaluates to `false`, then execute `skip`.

Gap 4. *Define/describe while loop statements* (**6 marks**)

Scoping Statements: A scoping statement introduces a new program scope, which is denoted by the `begin` and `end` tokens.

Sequential Composition: Sequential composition is used to execute an ordered list of valid statements. Each statement must be seperated by a semicolon ‘`;`’. Statements should be executed from top to bottom, and from left to right.

E.g. Suppose `A`, `B`, `C` are valid WACC statements. `A; B` means to first execute `A`, then `B` after `A` has finished.

- If expression `A` changed the state of the program, then `B` executes under the new state.
- If `A` does not terminate, then neither does `A; B`.
- If `A` terminates, then `B` starts, and `A; B` terminates only when `B` does.

Sequential composition has the following properties:

- `skip; A` and `A; skip` are both equivalent to just `A`.
- Sequential composition is associative: i.e. `A; B; C` may be read `(A; B); C` or `A; (B; C)`.

Gap 5. *Define/describe sequential composition i.e.* `<stat> ; <stat>` (**2 marks**)

Representation	ASCII Value	Description	Symbol
`\0`	`0x00`	null terminator	NUL
`\b`	`0x08`	backspace	BS
`\t`	`0x09`	horizontal tab	HT
`\n`	`0x0a`	line feed (new line)	LF
`\f`	`0x0c`	form feed (new page)	FF
`\r`	`0x0d`	carriage return	CR
`\"`	`0x22`	double quote	"
`\'`	`0x27`	single quote	'
`\\`	`0x5c`	backslash	\

Table 2: The escaped-characters available in the WACC language.

3.6 Expressions

A expression ⟨*expr*⟩ consists of a literal (integer, boolean, character, string or pair), a variable, an array element, a unary expression, a binary expression or an expression enclosed by parenthesis.

We discuss the meaning of each of these expressions in more detail below.

The expressions of the WACC language have been chosen to be side-effect free. This means that the evaluation of WACC expressions does not in any way change the program state or interact in any way with the 'outside world' (i.e. input and output are not allowed).

Gap 6. *Define side-effect free expressions* **(1 mark)**

Integer Literals: An integer literal ⟨*int-liter*⟩ consists of a sequence of decimal digits. Optionally, the sequence can be preceded by a `+` or a `-` symbol.

Boolean Literals: A boolean literal ⟨*bool-liter*⟩ is either `true` or `false`.

Character Literals: A character literal ⟨*char-liter*⟩ is a single ASCII character between two `'` symbols. A \ can be used to escape the character that immediately follows the \. The meaning of each escaped character is shown in Table **??**.

Gap 7. *Fill in Table* **??** **(2 marks)**

String Literals: A string literal ⟨*str-liter*⟩ is az sequence of characters between two `"` symbols. Each character in the string literal can be escaped in the same way as in character literal.

Pair Literals: The only pair literal ⟨*pair-liter*⟩ is `null` which represents a reference that does not point to any pair. To see how pairs are created, read the `newpair` case of the assignment statement.

Array Literals: Array literals cannot occur directly in expressions, but they do occur in the WACC language as assignment values. An array literal starts with a `[` token and ends with a `]` token. The elements of the array (zero or more) are given between these brackets and are separated by `,` tokens. All elements of an array must be of the same type, so the type of any non-empty array literal can be statically determined. If, however, an array literal is empty, we allow it to be of any array type. For example, the array `[]` can be of type `int[]`,`bool[]`, `char[]`, etc... depending on the context, but the array `[1]` must be of type `int[]`.

Variables: When a variable expression ⟨*ident*⟩ is evaluated it returns the value of that variable. If the variable is of type `T` then the return type of the expression is also `T`.

Operator	Argument Type	Return Type	Meaning
`!`	`bool`	`bool`	Logical Not
`-`	`int`	`int`	Negation
`len`	`T[]`	`int`	Array Length
`ord`	char	int	ASCII Number of Char
`chr`	int	char	Char from ASCII Number

Table 3: The unary operators of the WACC language with their types and meanings.

Array Elements: An array element expression evaluates to return an element from an array. The expression consists of two sub-expressions, the first of which must be of type `T[]` and the second of which must be of type `int`. The return type of the overall expression is `T`.

The first expression is evaluated to find an array `a` and the second is evaluated to find an index `i`. The overall expression returns the element at the index `i` of array `a`, that is, `a[i]`.

If the array has length l then the index `i` must be between 0 and $(l - 1)$, otherwise the expression will generate a runtime error.

Unary Operators: A unary operator ⟨*unary-oper*⟩ has a single sub-expression. The unary operators available in the WACC language are shown in Table ??. All unary operators have the same precedence, they are evaluated from right to left.

Gap 8. *Fill in Table* ?? **(2 marks)**

- The `!` operator performs a logical Not operation on the result of evaluating its sub-expression, returning `true` if the sub-expression evaluates to `false` and vice-versa.
- The `-` operator inverts the sign of the evaluation of its sub-expression.
- The `len` operator returns the length of the array referenced by the evaluation of its sub-expression.
- The `ord` operator returns the integer value ASCII code of the evaluated sub-expression.

Gap 9. *Define/describe the `ord` operator* **(1 mark)**

- The `chr` operator returns the corresponding character of the evaluated sub-expression according to the standard ASCII table.

Gap 10. *Define/describe the `chr` operator* **(1 mark)**

Binary Operators: A binary operator is used in in-fix style between two sub-expressions. The binary operators available in the WACC language are shown in Table ??. The operators have different precedences, as illustrated in the table, with 1 being the highest and 6 being the lowest.

Gap 11. *Fill in Table* ?? **(2 marks)**

- The `*`, `/`, `%`, `+` and `-` operators all have their standard mathematical behaviour, where integer underflow/overflow results in a runtime error. If the divisor of a division (`/`) or modulus (`%`) operator is evaluated to `0`, then this also results in a runtime error. The result of a division operation is positive if both its dividend and divisor have the same sign, and negative otherwise. The result of a modulus operation has the same sign as its dividend.
- The `>`, `>=`, `<` and `<=` operators perform a comparison test on the evaluations of their sub expressions. They accept expressions of type `int` or `char`, but both expressions must have the same type. The result is `true` if the comparison of the evaluated expressions is true. Otherwise, the result it `false`.

Operator	Precedence	Argument 1 Type	Argument 2 Type	Return Type	Meaning
`*`	1	`int`	`int`	`int`	Multiply
`/`	1	`int`	`int`	`int`	Divide
`%`	1	`int`	`int`	`int`	Modulus
`+`	2	`int`	`int`	`int`	Plus
`-`	2	`int`	`int`	`int`	Minus
`>`	3	`int` or `char`	`int` or `char` (same as arg 1)	`bool`	Greater Than
`>=`	3	`int` or `char`	`int` or `char` (same as arg 1)	`bool`	Greater Than or Equal
`<`	3	`int` or `char`	`int` or `char` (same as arg 1)	`bool`	Less Than
`<=`	3	`int` or `char`	`int` or `char` (same as arg 1)	`bool`	Less Than or Equal
`==`	4	any	any (same as arg 1)	`bool`	Equality
`!=`	4	any	any (same as arg 1)	`bool`	Inequality
`&&`	5	`bool`	`bool`	`bool`	Logical And
`||`	6	`bool`	`bool`	`bool`	Logical Or

Table 4: The binary operators of the WACC language, with their types and meanings.

- The `==` operator performs an equality test on the evaluations of its sub-expressions. It accepts any two expressions of the same type. When applied to expressions of type `int`, `bool` or `char`, the result is `true` iff the content of the two arguments are the same. When applied to expressions of type `T[]` or `pair`, the result is `true` iff the two references point to the same object Extension:Removal ofof the same type, as it has already been stated that they have to be of the same type, and if the reference points to the same object the type will be the same (due to how the language works). Otherwise, the result is `false`.
- The `!=` operator returns the opposite result to the `==` operator.
- The `&&` operator performs a logical And operation on the result of evaluating its sub-expressions, returning `true` if both sub-expressions evaluate to `true` and `false` otherwise.
- The `||` operator performs a logical Or operation on the result of evaluating its sub-expressions, returning `true` if either sub-expression evaluates to `true` and `false` otherwise.

Parenthesis: We can introduce a pair of parenthesis around an expression to control its evaluation. The expression in a parenthesis is always evaluated first, regardless of the operator precedence.

3.7 Whitespace and Comments

Whitespace is used in the WACC language to delimit keywords and variables. For example, `if a == 13` denotes the start of an `if` statement with boolean condition `a == 13`, whereas `ifa == 13` denotes a boolean expression comparing the variable `ifa` with the value `13`. Any other type of occurrence of whitespace is ignored by the compiler. Note, in particular, that the code indentation in the example programs has no meaning, it simply aids readability. Also note that whitespace inside a string or character literal is preserved by the compiler. Note that newlines are treated in the same fasion as whitespace (and so can be used as a delimiter in the same way), with the exception of for comments

Comments are not interpreted by the compiler, and can be used to provide further explanation of what a variable, function or statement does. Comments may only contain ASCII characters, and can be added anywhere in the source code.

A commnent is started using the `#` symbol, and is terminated at the following newline, hence no further code on the same line will be executed.

E.g.

- On a separate line:

  ```
  # Descriptive comment here
  int x = 10;
  ```

- In-line:

  ```
  int x = 10; # Descriptive comment here
  ```

Note that the '`#`' character may occur within a character or a string. In this case, the '`#`' character should not begin a comment.

Gap 12. *Define/describe comments* **(3 marks)**