The WACC Language Specification

Second Year Computing Laboratory
Department of Computing
Imperial College London

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.

This specification is carefully worded to ensure that the contents are non-ambiguous: ambiguity is the worst-enemy of a language designer. While there may be undefined behaviour in the language, the following keywords are given specific meanings that should be adhered to carefully throughout the implementation.

1. **MUST** This word, or the terms "**REQUIRED**" or "**SHALL**", denotes an absolute requirement of the specification.

2. **MUST NOT** This phrase, or the phrase "**SHALL NOT**", denotes an absolute prohibition of the specification.

3. **SHOULD** This word, or the adjective "**RECOMMENDED**", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

4. **SHOULD NOT** This phrase, or the phrase "**NOT RECOMMENDED**" mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.

5. **MAY** This word, or the adjective "**OPTIONAL**", mean that an item is truly optional.

Contents

1 WACC Language Syntax

The syntax of the WACC language is described in Backus-Naur Form (BNF) notation. To make the presentation more precise, the following extensions are provided to BNF notation:

- (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.

Definition 1 *Two grammatical constructs* x *and* y *are said to be* juxtaposed *when separated by whitespace, for example* x y.

1.1 Lexical Structure

The syntax of WACC is formed up of lexical tokens. These tokens are all indivisible, and whitespace (including comments) **MUST NOT** occur within any token. Tokens **SHOULD** be parsed according to "longest-match", such that the longest continuous matching token is recognised after the end of the previous token. Within this sub-section, juxtaposition in the grammar does not implicitly allow for whitespace.

⟨*int-liter*⟩ ::= ⟨*int-sign*⟩? ⟨*digit*⟩+

⟨*digit*⟩ ::= ('0'-'9')

⟨*int-sign*⟩ ::= '+' | '-'

⟨*bool-liter*⟩ ::= 'true' | 'false'

⟨*char-liter*⟩ ::= ''' ⟨*character*⟩ '''

⟨*str-liter*⟩ ::= '"' ⟨*character*⟩* '"'

⟨*character*⟩ ::= any-graphic-ASCII-character-except-'\'-'''-'"' (graphic g ≥ ' ')
| '\' ⟨*escaped-char*⟩

⟨*escaped-char*⟩ ::= '0' | 'b' | 't' | 'n' | 'f' | 'r' | '"' | ''' | '\'

⟨*pair-liter*⟩ ::= 'null'

⟨*ident*⟩ ::= ('_' | 'a'-'z' | 'A'-'Z') ('_' | 'a'-'z' | 'A'-'Z' | '0'-'9')* (not keyword)

⟨*comment*⟩ ::= '#' (any-character-except-EOL)* (⟨*EOL*⟩ | ⟨*EOF*⟩)

Further to the above described tokens, which may appear in the subsequent sub-sections, any literal string from this point, for instance 'if', will be treated as a token in its own right. Whitespace and comments, which can be found between tokens, have no semantic meaning within the program. From this point, there is implicit optional whitespace between the juxtaposition of terminals, non-terminals, and symbols.

Precedence	Associativity	Operators
0 (tightest)	prefix	'`!`', '`-`', '`len`', '`ord`', '`chr`'
1	infix left	'`*`', '`%`', '`/`'
2	infix left	'`+`', '`-`'
3	infix non	'`>`', '`>=`', '`<`', '`<=`'
4	infix non	'`==`', '`!=`'
5	infix right	'`&&`'
6 (weakest)	infix right	'`||`'

Table 1: The precedence of operators in the WACC language.

1.2 Expressions

Expressions consist of literals, parenthesised expressions and array indexing, separated by unary and binary operations. For simplicity of grammar presentation, binary and unary operations have been coalesced; in practice all operators are assigned a precedence and associativity.

⟨*expr*⟩	::=	⟨*unary-oper*⟩ ⟨*expr*⟩
	|	⟨*expr*⟩ ⟨*binary-oper*⟩ ⟨*expr*⟩
	|	⟨*atom*⟩
⟨*atom*⟩	::=	⟨*int-liter*⟩
	|	⟨*bool-liter*⟩
	|	⟨*char-liter*⟩
	|	⟨*str-liter*⟩
	|	⟨*pair-liter*⟩
	|	⟨*ident*⟩
	|	⟨*array-elem*⟩
	|	'`(`' ⟨*expr*⟩ '`)`'
⟨*unary-oper*⟩	::=	'`!`' | '`-`' | '`len`' | '`ord`' | '`chr`'
⟨*binary-oper*⟩	::=	'`*`' | '`/`' | '`%`' | '`+`' | '`-`' | '`>`' | '`>=`' | '`<`' | '`<=`' | '`==`' | '`!=`' | '`&&`' | '`||`'
⟨*array-elem*⟩	::=	⟨*ident*⟩ ('`[`' ⟨*expr*⟩ '`]`')+

The precedence and associativities of the operators are described in table 1. The precedence ordering of the operators **MUST** be respected. A left-associative operator **MUST** be able to appear repeatedly and **MUST** implicitly bracket to the left, for instance, '`5 + 6 + 7`' and '`(5 + 6) + 7`' represent the same expression. A right-associative operator **MUST** be able to appear repeatedly and **MUST** implicitly bracket to the right, for instance, '`p && q && r`' and '`p && (q && r)`' represent the same expressions. A non-associative operator **SHOULD NOT** appear repeatedly, and therefore does not bracket – for example, '`x < y < z`' **SHOULD NOT** be syntactically legal.

Brackets have no semantic meaning, they just override syntactic precedence.

1.3 Types

The syntactic description of types within the WACC language is given below:

⟨*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`’

Importantly, non-erased pair types **MUST NOT** appear directly within another non-erased pair, the rationale for this is discussed in section 2.1.3. However, note that array types **MUST** be able to interleave with pair types, such that ‘`pair(int, pair(int, char)[])`’ is syntactically (and semantically) legal.

1.4 Statements

WACC, as described in this specification, is a procedural language that is statement-oriented. Statements are described as follows:

⟨*program*⟩ ::= ‘`begin`’ ⟨*func*⟩* ⟨*stmt*⟩ ‘`end`’

⟨*func*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩ ‘`(`’ ⟨*param-list*⟩? ‘`)`’ ‘`is`’ ⟨*stmt*⟩ ‘`end`’

⟨*param-list*⟩ ::= ⟨*param*⟩ (‘`,`’ ⟨*param*⟩)*

⟨*param*⟩ ::= ⟨*type*⟩ ⟨*ident*⟩

⟨*stmt*⟩ ::= ‘`skip`’
| ⟨*type*⟩ ⟨*ident*⟩ ‘`=`’ ⟨*rvalue*⟩
| ⟨*lvalue*⟩ ‘`=`’ ⟨*rvalue*⟩
| ‘`read`’ ⟨*lvalue*⟩
| ‘`free`’ ⟨*expr*⟩
| ‘`return`’ ⟨*expr*⟩
| ‘`exit`’ ⟨*expr*⟩
| ‘`print`’ ⟨*expr*⟩
| ‘`println`’ ⟨*expr*⟩
| ‘`if`’ ⟨*expr*⟩ ‘`then`’ ⟨*stmt*⟩ ‘`else`’ ⟨*stmt*⟩ ‘`fi`’
| ‘`while`’ ⟨*expr*⟩ ‘`do`’ ⟨*stmt*⟩ ‘`done`’
| ‘`begin`’ ⟨*stmt*⟩ ‘`end`’
| ⟨*stmt*⟩ ‘`;`’ ⟨*stmt*⟩

⟨*lvalue*⟩ ::= ⟨*ident*⟩ | ⟨*array-elem*⟩ | ⟨*pair-elem*⟩

⟨*rvalue*⟩ ::= ⟨*expr*⟩
| ⟨*array-liter*⟩
| ‘`newpair`’ ‘(’ ⟨*expr*⟩ ‘`,`’ ⟨*expr*⟩ ‘)’
| ⟨*pair-elem*⟩
| ‘`call`’ ⟨*ident*⟩ ‘(’ ⟨*arg-list*⟩? ‘)’

⟨*arg-list*⟩ ::= ⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)*

⟨*pair-elem*⟩ ::= ‘`fst`’ ⟨*lvalue*⟩ | ‘`snd`’ ⟨*lvalue*⟩

⟨*array-liter*⟩ ::= ‘`[`’ (⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)*)? ‘`]`’

Note that ⟨*stmt*⟩ allows multiple statements to be delimited by semi-colons. The statement delimiter, ‘`;`’, is treated as an infix operator with an unspecified associativity – a trailing ‘`;`’ **SHALL NOT** be legal. There is no parenthesisation allowed of sequenced statements.

A WACC file (extension `.wacc`) only ever contains a single WACC program.

1.5 Additional Syntactic Restrictions

All errors generated as a result of a failure for a user program to adhere to the synactic rules described in sections 1.1 to 1.4 **MUST** be denoted as *syntax errors*. However, in addition to these, the violation of the following two requirements **MUST** also be reported as syntax errors.

- Integer literals, ⟨*int-liter*⟩, **MUST** be within the range of a signed 32-bit decimal number. The maximum possible value is $2^{31} - 1$ and the minimum possible value is -2^{31}.

- It is **REQUIRED** that all functions, ⟨*func*⟩, can only be exited via a ‘`return`’ or ‘`exit`’ statement. There **MUST NOT** be any code following the *last* ‘`return`’ or ‘`exit`’ of any execution path. Not all execution paths through a function need converge and pass through the *same* ‘`return`’ or ‘`exit`’, however.

2 WACC Language Semantics

The semantics of the WACC language are split into typing, scoping, and behaviour.

2.1 Types

The WACC language is *strongly* and *statically* typed. This means that, with some noted exceptions:

- Each variable's type can be determined at compile-time (statically) from its declaration site.
- Types are specific and cannot be freely coerced between, nor is there any casting permitted.

In short, the exceptions to these rules are that nested pair types are subject to erasure and there is a permitted weakening of '`char[]`' into '`string`'. Throughout this sub-section, types such as τ and σ denote arbitrary types.

Any programs that are ill-typed do not compile, and **MUST** produce a *semantic error*. Valid implementations **MAY** choose to give errors arising from ill-typed programs more specific names.

2.1.1 Basic Types

The basic types in the WACC language are:

- `int`: a 32-bit signed integer type.
- `bool`: the two valued boolean type consisting of `true` and `false`.
- `char`: 7-bit ASCII character.
- `string`: a (statically) immutable sequence of characters.

2.1.2 Arrays

An array of elements of type τ is denoted by 'τ`[]`'. The element type τ can be any type except for an erased '`pair`' (see section 2.1.3). The length of the array **MUST** be fixed at its creation, and it **MUST** be heap allocated. Arrays **MUST** track their size *dynamically* – the size of an array **MUST NOT** be tracked only at compile-time.

String weakening The type `char[]` may take the place of `string` for the purposes of type-checking. Such arrays **MUST NOT** be frozen and can still be modified by the programmer. It is the programmer's responsiblity to retain a reference to the original array so that it can be freed – the '`free`' statement cannot be used with `string`. This is a uni-directional type coercion: a `string` **MUST NOT** take the place of `char[]`, as a `string` may not be modified.

Definition 2 *A type τ is* compatible *with another type σ if and only if $\tau \equiv \sigma$ or τ can be weakened to σ.*

Definition 3 *The* lowest common ancestor *σ of a set of types $\{\tau_1, .., \tau_n\}$ is the most specific type such that each of τ_1 through τ_n is compatible with σ.*

Invariant As arrays are mutable, they are necessarily *invariant* in their type parameter. This means that `char[][]` **MUST NOT** be compatible with `string[]`. Were this the case, it would allow a dangerous mutable view on `string`. However, it remains possible to store a `char[]` into a `string[]` by the regular weakening relation – this is not unsound.

2.1.3 Pairs

Pair types in WACC are binary tuples that can be used to build up any other datastructure. Given two (possibly distinct) types, τ and σ, the type ‘`pair(`τ`, `σ`)`’ denotes a pair whose first element has type τ and second element has type σ. The literal ‘`null`’ has type `pair(`τ`, `σ`)` for all types τ and σ.

Erasure When either of τ or σ within a pair are themselves a pair, they **MUST** have an *erased* type ‘`pair`’. For instance, ‘`pair(int, pair)`’ is legal, but ‘`pair(int, pair(int, bool))`’ is not. Note that single-element arrays can be used to losslessly nest pairs. `pair(`τ`, `σ`)` can be weakened to `pair` **and** `pair` can be weakened to `pair(`τ`, `σ`)`, in other words `pair` and `pair(`τ`, `σ`)` are fully coercible.

The implication of the nested-pair erasure is that some typing information is necessarily lost, but this allows for recursively constructed datastructures arising from nested-pairs: if nested pairs were not subject to erasure, this would involve infinite types.

A consequence of erasure is that WACC’s type system is *unsound*: it is possible to perform unsafe coercions between types by storing and extracting them from a nested-pair. Neither the runtime or compiler are required to prevent these coercions, and using an improperly coerced value is **undefined behaviour**.

Since the type `pair` is not itself a legal type of a variable, nested pairs will cease to be erased once they are extracted; at this point, the full type of the pair **MUST** be known – this is discussed in section 2.1.4.

Invariant As a mutable type, pairs are also necessarily *invariant* in their type parameters. This means that ‘`pair(char[], int)`’ **MUST NOT** be compatible with the type ‘`pair(string, int)`’. Similarly to arrays, it is still possible to assign a `char[]` to the left-hand argument of `pair(string, int)`, as this does not create an unsoundness.

2.1.4 Expressions and Statements

Array literals Array literals can appear on the right-hand side of assignments. The set of types of all elements in the array **MUST** have a lowest common ancestor τ; the type of the literal **SHALL** then be τ`[]`. For instance, an array literal containing a mix of `char[]` and `string` **SHALL** have overall type `string[]`. An empty array literal ‘`[]`’ has type τ`[]` with no specifically discernable type.

Operators Each of the binary and unary operators within WACC work on a specific set of types, as illustrated in tables 2 and 3. Equality is well defined for all types, so long as the two operands have the same type: `int[]` and `char[]` are not comparable with each other, even though they are both arrays. Comparison can only be performed between arguments of matching types as well.

Operator	Argument Type	Return Type
‘`!`’	`bool`	`bool`
‘`-`’	`int`	`int`
‘`len`’	τ`[]`	`int`
‘`ord`’	`char`	`int`
‘`chr`’	`int`	`char`

Table 2: The types of unary operators in WACC.

Operators	Left-Hand Type	Right-Hand Type	Return Type
‘`*`’, ‘`/`’, ‘`%`’, ‘`+`’, ‘`-`’	`int`	`int`	`int`
‘`>`’, ‘`>=`’, ‘`<`’, ‘`<=`’	`int/char`	`int/char`	`bool`
‘`==`’, ‘`!=`’	τ	τ	`bool`
‘`&&`’, ‘`||`’	`bool`	`bool`	`bool`

Table 3: The types of binary operators in WACC.

Return statements The type of values provided to a return statement **MUST** be compatible with the return type of the enclosing function – since the main scope has no return type, ‘`return`’ **SHALL NOT** be legal within the main body of the program, this is a *semantic error*.

Printing Print statements may print values of any type, though the behaviour of the printing may vary – this is discussed in section 2.3.6.

Reading The ‘`read`’ statement takes a destination ⟨*lvalue*⟩ which must be of either type `int` or type `char`. No other types of values may be read.

Exit statements The ‘`exit`’ statement must be provided with an exit code of type `int`.

Memory freeing The ‘`free`’ statement takes an argument that must be of type τ`[]` or `pair(`τ`, `σ`)`.

Control-flow statements Both ‘`if`’ and ‘`while`’ constructs have a condition of type `bool`.

Declaration and assignments When a variable of type τ is declared, the right-hand side of the assignment **MUST** be compatible with the type τ. A variable's type **MUST** be fixed at its declaration site. Similarly, the right-hand side of a non-declaration assignment **MUST** be compatible with the left-hand side. The right-hand sides of assignments are ⟨*rvalue*⟩s, which includes expressions, as well as array literals, pair construction, pair extraction, and function calls.

A function call's return type **MUST** be compatible with the left-hand type of the assignment and the call **MUST** be fully saturated, provided with all its arguments, and every argument **MUST** be compatible with the corresponding parameter in the function's declaration.

Given two expressions e_1 of type τ and e_2 of type σ, ‘`newpair(`e_1`, `e_2`)`’ will have type `pair(`τ`, `σ`)`.

Definition 4 *A type* κ *is* known *if it is compatible with* τ[], `pair(`τ`, `σ`)`, `int`, `char`, `bool`, *or* `string`.

Both ⟨*lvalue*⟩s and ⟨*rvalue*⟩s can represent (nested) pair extraction. This has some more complex typing rules that **MUST** be carefully followed:

- The type of '`fst p`', where '`p`' has type `pair(`τ`, `σ`)`, is τ. An update to '`fst p`' **MUST** be compatible with τ, and an extraction from '`fst p`' **MUST** ensure τ is compatible with the left-hand side of the assignment.
- The type of '`snd p`', where '`p`' has type `pair(`τ`, `σ`)`, is σ. An update to '`snd p`' **MUST** be compatible with σ, and an extraction from '`fst p`' **MUST** ensure σ is compatible with the left-hand side of the assignment.
- When considering a *nested* pair extraction or update, i.e. '`fst fst p`', where '`p`' necessarily has type `pair(pair, `τ`)`, the type of the extractee is not known as it comes from `pair`.

  When '`p`' has type `pair`, the type of '`fst p`' or '`snd p`' as either an update or extraction is instead a known type κ determined by the other side of the assignment. Any unsoundness caused by type coercion is the fault of the programmer.

  If the type of the other side of the assignment is not known, then this **MUST** result in a type error. For example '`fst fst p = fst fst q`' **MUST** raise an error, since neither side of the assignment is of some known type κ.

2.2 Scoping

In WACC, scopes are introduced by '`begin .. end`', functions, if-statements, and while-loops. The top-level global scope contains all other definitions within the program.

Definition 5 *A* binding *is a name associated with either a variable or a function argument.*

Definition 6 Lexical *(or* static*) scoping is where variables are in scope based on their position within the source file, and not on the dynamic execution path of the program.*

2.2.1 Variable Declarations

Variables are always declared within a scope, and **MUST** be unique within that scope. The lifetime of any variable is static: all uses of a variable **MUST** appear below the declaration site. A variable **MUST** be assigned its type at declaration. When a scope is exited, any variables declared within **MUST** no longer be accessible. A variable **MUST NOT** be in scope in the right-hand side of its own declaration.

2.2.2 Nested Scopes

A scope may be created inside another scope. The child scope **MUST** inherit all bindings from the parent scope – this cascades, so bindings are accessible from grandparent scopes etc. When a scope declares a variable with the same name as a binding from an enclosing scope, it **MUST** *shadow* that binding so that it cannot be referred to from that point until the new variable goes out of scope. As a consequence, a statement such as '`int x = x`' may be legal, so long as '`x`' is declared in an enclosing scope (this is to be read as '`int` x_2 `=` x_1'). A shadowed variable **MUST** be able to have a different type to the pre-existing bindings.

2.2.3 Functions

Functions can only be declared at the start of the main scope (as demanded by the syntax of Wacc). A function body's parent scope **MUST** contain the bindings for each argument of the function. The body then follows the scoping rules of section 2.2.2. As functions are declared within the global scope, they **SHALL** be accessible everywhere, and are allowed to referred to before they are lexically defined: this allows for recursive as well as mutually recursive functions.

In Wacc, functions are not first-class values in the language. They can only be called unambiguously via a '`call`' statement. As such, variables and functions may share the same names without conflict: for exmaple, '`int foo(bool foo) is return 0 end`' is a legal function.

2.3 Behaviour

This section describes the operational behaviour of the constructs within the Wacc language. All behaviours assume a syntactically-valid, well-typed, and well-scoped program.

2.3.1 Runtime Representations

Each of the runtime values in Wacc **MUST** be able to accommodate the following values:

- `int` **MUST** support the full range of values between -2^{31} and $2^{31} - 1$. No runtime value outside of this range is expressible within the program.
- `bool` **MAY** be as small as 1-bit wide.
- `char` **MUST** support 7-bit ascii.
- `string`s and arrays **SHOULD** share a common representation (to allow for easy conversion). They **MUST** be represented as pointers and track their number of characters/elements in their allocated data. While arrays **MUST** be heap-allocated, strings **SHOULD** be stored within the data-segment of the executable to avoid memory leaks. While both the empty string and empty array are valid values, there is no such "null" value for these types. Arrays **MUST** be explicitly deallocated with '`free`'; it is the programmer's responsibility to ensure no memory-leaks.
- Pairs **MUST** be represented as pointers. As unsafe pair coercion is **undefined behaviour**, and pair extraction is only legal on a *known* type (which each have a statically known size), it is possible to have the representation of the elements of a pair vary based on type. Pairs are mutable, and **MUST** be allocated on the heap; like arrays, pairs can be deallocated with a '`free`' statement. The '`null`' value does exist for pairs, it is not allocated on the heap.

The runtime representations of these types **MAY** be represented by wider sizes, however they **MUST** remain within the stated value ranges from the programmer's point-of-view. The size of a pointer is platform dependent. The runtime representation can change as convenient for the current storage – for instance, it might not be feasible to store each value in less than 64-bits when occupying a register, but **SHOULD** be more space efficient when in memory.

2.3.2 Expressions

The expressions of the WACC language have been chosen to be side-effect free. This means that the process of evaluating an expression does not change the program state, though they can throw runtime errors. They are evaluated as follows:

Literals All literals within WACC expressions plainly represent themselves and are already evaluated.

Variables The evaluation of a variable simply returns the value stored in that variable.

Array Index Evaluation of an array index '`arr[i]`' proceeds as follows:

1. evaluate the expression '`arr`' to obtain a pointer to the array, p
2. evaluating the index '`i`' obtaining an integer n
3. checking that $0 \leq n < \text{length}(p)$, if not a runtime error **MUST** be thrown
4. return the the n^{th} element in the array p

Unary Operators A unary operator has a single sub-expression, which is evaluated to obtain a value, x, of the correct type. Then:

- '`!`' performs a logical negation of x, such that '`true`' becomes '`false`', and vice-versa.
- '`-`' behaves as $0 - x$, inverting the sign of the value x. If any overflow or underflow occurs, this **MUST** result in a runtime error.
- '`len`' returns the length of the array pointed to by x. This **MUST** be computed in $O(1)$.
- '`ord`' converts the character x into its ASCII ordinal value. For example '`ord 'a'`' is '`97`'.
- '`chr`' converts the integer x into the corresponding ASCII character for that ordinal. It **MUST** be true that $0 \leq x \leq 127$, if this is not the case, a runtime error **MUST** be raised.

Binary Operators A binary operator has two sub-expressions, which are evaluated, in either order, to obtain values x and y of the correct types. Then:

- The '`*`', '`+`' and '`-`' operators all have their standard mathematical behaviour on integers. Any operation that results in overflow or underflow **MUST** throw a runtime error.
- Both '`/`', '`%`' must both first check that the divisor y is not 0 – if it is they **MUST** throw a runtime error. They both have their standard mathematical meanings. For division, if both x and y have the same sign, the result is positive and negative otherwise. For modulus, the sign of the result is the same as that of the dividend x. Overflow in division is left as **undefined behaviour** and throwing a runtime error is not required.
- The '`>`', '`>=`', '`<`' and '`<=`' operators all perform a standard comparison on x and y.

- The '`==`' operator compares x and y for equality. For `int`, `char`, `bool` this standard value equality; for τ`[]`, `string`, and `pair(`τ`, `σ`)`, however, this **MUST** be a referential equality on the pointers.
- The '`!=`' operator behaves like '`!(`x `==` y`)`'.
- The '`&&`' operator evaluates to `true` when both x and y are `true`, and `false` otherwise. This operation **MAY** short-circuit and only evaluate x if it is `false`.
- The '`||`' operator evaluates to `false` when both x and y are `false`, and `true` otherwise. This operation **MAY** short-circuit and only evaluate x if it is `true`.

2.3.3 Function Definitions

As noted in section 2.2.3, the execution of a function runs in an empty lexical scope with a parent containing the bindings for the arguments. WACC is an eager pass-by-value language, as such, all function arguments **MUST** be evaluated before being passed into the function; the order-of-execution is **not specified**, since expression evaluation does not have side-effects.

Note that since pairs, arrays, and strings **MUST** all be represented as pointers, any mutable updates **SHALL** be reflected outside the function call: this is **not** pass-by-reference, as a reassignment to an argument **SHALL NOT** be reflected in the caller's scope.

Return statements A '`return`' statement can only be present in the body of a function; when executed, it will evaluate its argument, and pass the resulting value back to the caller of the function, exiting the current call immediately.

2.3.4 Sequentialisation and No-ops

The '`skip`' statement has no effect on the program when executed. Given 'S_1 `;` S_2', first, the statement S_1 is executed, and then the statement S_2 is executed, observing any changes that S_1 may have made during execution. Note that '`skip ;` S' and 'S `; skip`' **MUST** both be semantically equivalent to 'S'.

The '`skip`' statement can be used to ignore unused branches of conditional statements, for instance.

2.3.5 Assignments and Declarations

Both regular assignments and declarations have the same runtime behaviours when considering their right-hand sides. Each case is explained below, with each case producing a single value referred to as x.

⟨*rvalue*⟩ evaluation

- A plain expression **SHALL** be evaluated into the value x.
- An array literal is evaluated into a value as follows:
    1. allocate enough memory on the heap to store all elements and the size of the array; if allocation fails, this **MAY** produce a runtime error
    2. evaluate all elements of the array (the order-of-evaluation is unspecified)
    3. store each evaluated element into its corresponding slot in the array

Steps (2) and (3) can be interleaved, or done sequentially. The value x **SHALL** be the pointer returned by the memory allocation.

- A function call proceeds as outlined in section 2.3.3. The return value of the call **SHALL** be x.
- A '`newpair`' will:
  1. allocate enough memory on the heap to store the pair (see section 2.3.1); if allocation fails, this **MAY** produce a runtime error
  2. evaluate both provided expressions (order-of-evaluation is unspecified)
  3. store each evaluated element into its corresponding slot in the pair

  Steps (2) and (3) can be interleaved, or done sequentially. The value x **SHALL** be the pointer returned by the memory allocation.
- A pair extraction ('`fst p`' or '`snd p`') is evaluated into a value by:
  1. obtaining the *reference*, r, to the location of the pair '`p`' according to the rules outlined for ⟨*lvalue*⟩s
  2. *following* r to obtain a pointer, p, to the pair
  3. if p is the `null` pair, this **MUST** throw a runtime error
  4. extracting the first or second element of p respectively, according to its underlying runtime representation

  The extracted element **SHALL** be the value x.

Definition 7 *A* reference *describes a location, which could either be a specific register, a stack address, or a heap pointer. It has no specific representation, and serves to denote a place that can be* followed *or stored into. References cannot be nullary.*

Definition 8 *A reference can be* followed *to obtain the value that exists at that location. For a register, this means refering directly to that register; or for a memory address this means dereferencing the pointer.*

⟨*lvalue*⟩ **evaluation** Obtaining a reference, r, to the location represented by an ⟨*lvalue*⟩ requires evaluating the term until a concrete location is obtained.

- an identifier '`v`' is already a reference, r.
- a reference to an array index '`arr[i]`' is obtained by:
  1. determining the reference, $r_{\texttt{arr}}$, to the location of the array '`arr`'
  2. following $r_{\texttt{arr}}$ to obtain a pointer representing the array, p
  3. evaluating the index '`i`' obtaining an integer n
  4. checking that $0 \leq n < \text{length}(p)$, if not a runtime error **MUST** be thrown
  5. r represents the address of the n^{th} element in the array p

The evaluation of the array and index can happen in any order.

- a reference to a pair element '`fst p`' or '`snd p`' is obtained by:
  1. determining the reference $r_{\texttt{p}}$, to the location of the pair '`p`'
  2. following $r_{\texttt{p}}$ to obtain a pointer to the pair, p
  3. if p is the `null` pair, this **MUST** throw a runtime error
  4. r represents the address of the first or second element in the pair p

Given a value to store, x, and reference, r, an assignment will write the value x into the location represented by r.

2.3.6 IO Statements

WACC can interact with the outside world in two ways: reading input from and printing to the terminal.

Read Statements A read statement '`read`' extracts a "token" from the input stream `stdin` and stores it in a given location. The input stream will consist of space-separated characters and integers. First, a reference r is obtained from the given ⟨*rvalue*⟩ (as described in section 2.3.5); then a value x is read from the input stream according to the following rules:

- If the type of the read is `char`, a single ASCII character is read from the input.
- If the type of the read is `int`, a whole integer is taken from the input stream, which may consist of many characters.

If no input can be read, the `read` statement **MUST** do nothing and continue, otherwise the value x read from the stream is written into the location represented by r. If the read did fail, then the value stored at r **MUST NOT** have been modified.

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`	Outputs the character, respecting the special meanings in table 5.	`'c'`	"c"
`string` or `char[]`	Output each character in order, respecting the special meanings in table 5.	`"hello"`	"hello"
Other Array Types	Output the start address of the array in memory in hexadecimal.	N/A	"0x12008"
`pair`	Output the start address of the pair in memory in hexadecimal. A '`null`' **MUST** print "(nil)" instead.	N/A	"0x13458"

Table 4: The behaviour of the print statements for each type of expression.

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

Table 5: The escape-characters available in the WACC language.

Print Statements The two print statements, ‘`print`’ and ‘`println`’ behave similarly, except that ‘`println`’ outputs an additional newline at the end of the print. The output representation of a printed expression depends its type, this is summarised in table 4.

2.3.7 Control-Flow Statements

WACC has five statements for controlling the flow of a program’s execution. Function calls and returns have been covered in section 2.3.3; the others are conditional statements – often called ‘`if`’-statements, ‘`while`’-loops, and ‘`exit`’-statements.

Conditionals A conditional statement, ‘`if` c `then` S_1 `else` S_2 `fi`’, must first evaluate the expression c; if c evaluates to `true`, then S_1 is executed, otherwise S_2 is executed.

Looping A while loop, ‘`while` c `do` S `done`’, first evaluates the expression c; if c evaluates to `true`, then S is executed and the loop repeats with the updated program state. When c no longer evaluates to `true`, the statement S is not executed again and the loop finishes.

Exit statements An exit statement ‘`exit`’ terminates the program. The given expression is first evaluated to provide an integer, and the 8 least-significant bits are taken as the exit code. The program is terminated immediately by calling the `exit` system call with the computed code.

2.3.8 Freeing Memory

The ‘`free`’ statement is used to deallocate heap memory allocated by arrays and pairs. For nested structures, only the outer-most pointer is deallocated, meaning inner pointers can leak unless they have already been freed. Using a deallocated pointer after freeing is **undefined behaviour**. When considering pairs, attempting to deallocate the `null` value **MUST** produce a runtime error. Any memory leaks or dead pointers are the programmer’s responsibility.