Wybe: Design of a Programming Language

Peter Schachte *

Posted Date: 5 September 2025

doi: 10.20944/preprints202509.0517.v1

Keywords: programming languages; programming language design; declarative programming; imperative programming

 doi:10.20944/preprints202509.0517.v1

Article

Wybe: Design of a Programming Language

Peter Schachte

The University of Melbourne, Victoria 3010, Australia; schachte@unimelb.edu.au

Abstract

We propose a set of design principles to guide the design of a programming language intended for general, practical use. These principles centre around supporting development of robust programs, supporting independent development and evolution of separate components of an application, and developing programs with adequate performance. We identify one key principle, interface integrity, as the most important characteristic of declarative programming languages. Following these principles has led to the development of the WYBE programming language, which provides a range of features common in functional, procedural, and logic programming languages. In particular, we argue that it provides much of the benefit of declarative programming languages, while providing much of the flexibility of imperative programming.

Keywords: programming languages; programming language design; declarative programming; imperative programming

1. Introduction

WYBE[1] is a new programming language under development as a platform for research into programming language design. The goal of this research is to develop a programming language suitable for a wide spectrum of practical programming tasks, ranging from small to large scale, over a range of application domains. This section presents the principles we believe are most important for a wide spectrum programming language, which have guided the design of the language.

1.1. Correctness First

The primary goal guiding the design of WYBE is to make it easy to develop a program that does what is intended, without making it likely that the program does something unintended. Too much focus on the former half of this goal leads to a language that facilitates quick development of buggy software, while overemphasis of the latter half leads to a language that is unproductive or difficult to use. To balance these two aspects, we follow these principles:

Predictable behaviour: abstractions provided by the language and libraries, and how they interact, should behave in ways the programmer would expect. A programming language should be designed to enable and encourage programmers to produce abstractions that behave as expected. Because complexity of abstractions and their interactions easily confounds expectations, a corollary of this principle is that abstractions should be as simple and orthogonal as possible.

Expressive abstractions: the language should permit concise and precise expression of the programmer's intent.

Enforced consistency: the language should ensure that the program is internally consistent and consistent with the language's requirements. The program should not be permitted to run if it does not satisfy

[1] Pronounced "WEE-buh", WYBE is named for Edsgar Wybe Dijkstra, a pioneer in the areas of software engineering and programming language design.

 doi:10.20944/preprints202509.0517.v1

this requirement. This may conflict with the natural desire to produce programs quickly, but leads to more robust programs in the long run.

Robust abstractions: the operations of the language and libraries should fail in graceful ways when their preconditions are not met. Ideally, the language should ensure that preconditions are always met, or that the programmer has ensured that they are met. When this is infeasible or impractical, operations should fail in a way that prevents damaging outcomes, insofar as they can be anticipated.

Interface integrity: all data flow in a program should happen through well-defined interfaces. This is a central concept underpinning WYBE, and will be discussed in detail in Section 2.

1.2. Maintainability and Evolvability

Maintenance is a significant part of the software development process, often accounting for more than half of the total cost of a software project [1], therefore it is important that a programming language supports software evolution. For software developed with an agile methodology, maintenance is integrated throughout the development process, so the language's support for the evolution of a design is of even greater importance. Thus, a programming language should make it easy to write code that is malleable while remaining robust, leading to the following principles:

Modularity: Stevens *et al.* [2] identify the modularity of a program as a key factor in its maintainability. In particular, they identify two important characteristics related to modularity: *coupling* is a measure of how interdependent different components of a program are, and *cohesion* is a measure of how closely related the responsibilities of the components of a module are. They cite low coupling and high cohesion as central to program maintainability. The lowest level of coupling is called *data coupling*, in which all and only the necessary data is passed through parameters; this is closely related to the idea of interface integrity. A programming language can aid in this by prohibiting the worst forms of coupling, and by providing constructs to support cohesive modules.

Separation of concerns [3]: a program should be modularised into distinct parts, each of which has a clearly defined *interface* and a separate *implementation*. The interface is the concern of the users of the module, who need not worry about how the module is implemented, while the implementation is the concern of the programmer implementing the module, who need not worry about how it is used. There is value in separating these concerns, even when the user and implementer are the same person, as it allows the programmer to focus on one aspect at a time. This separation holds at the level of the module, as well as at the level of the individual operations of each module.

Data encapsulation and information hiding [4,5]: the operations on a data structure should be defined together with the data structure itself as a single unit. Furthermore, only the necessary operations should be visible to the users of the data structure; other operations should be hidden. Data encapsulation and information hiding support development of highly cohesive modules. Operations that are hidden can be changed or removed without affecting the users of the data structure, keeping the impact of such changes localised.

Irrelevance insensitivity: programmers should not be forced to make decisions that are unimportant to the meaning of the program but are difficult to change later. For example, most programming languages allow programmers to name the fields or members of a data structure, and use those names to access the data. This makes the code insensitive to the layout of the data structure, allowing it to be changed without needing to change code that accesses the data. A programming language should also make it easy to make decisions in such a way that they can later be changed with minimal impact on the rest of the program. A special case of this is the uniform reference principle [6], which states that accessing parts of a data structure should look the same as applying a function to the data, so that the programmer can change between storing and computing aspects of the data without needing to

 doi:10.20944/preprints202509.0517.v1

change code that uses it. A corollary of this principle is that the mutation or creation of data should look like any other operation, so that the programmer can change the way it is modified or produced without needing to change code that uses it.

1.3. Performance

To be suitable for a wide spectrum of applications, a programming language must deliver acceptable execution performance. Properly speaking, program performance is related to a programming language *implementation*, and not to the language itself. We discuss it here because the design of a programming language has a significant impact on the performance a programming language implementation can deliver. To support performance, we identify these principles:

Native code compilation: The compiler should produce executable machine code, rather than emulating an abstract machine, as that generally delivers the best performance. Additionally, language constructs should be designed to map well to machine code, avoiding constructs that are difficult to compile efficiently.

Optimising compilation: The compiler should do its best to optimise the generated code, taking advantage of information it can infer about the program.

Predictable performance: the execution model of the language should be simple enough that the programmer can roughly anticipate the time and space performance of the code they write. This suggests a preference for compiler optimisations whose applicability and effect are predictable. In most contexts, programmers will happily accept any unexpected performance improvement, so the desire for predictability should not preclude unpredictable performance improvements, as long as programmers are not led to expect optimisations that do not apply.

Incremental compilation: The ability to quickly move from editing source code to testing it is important for programmer productivity. This is particularly true after the first compilation and testing of a program, when small changes are made and need to be tested.

2. Interface Integrity

Interface integrity [7] is a characteristic of a program in which all data flow in the program passes through explicit interfaces. More precisely, we can think of every *execution unit* (*i.e.*, function, method, procedure, subroutine, or whatever user-defined operations are called) in a program as having an *effective interface*, which describes all the data that an individual execution of the execution unit uses but does not compute itself from other data, plus all the data that it produces and makes available outside that execution. This includes information passed to it by other execution units, or that it accesses from the file system or other input devices, or that it reads from global or static variables. Even where data was computed by an earlier execution of that execution unit, it is considered to come from outside the current execution, so it is part of its effective interface. It also includes information that it returns to other execution units, or that it writes into global or static variables, or that it writes to the file system or other output devices. Note that the effective interface reflects *what* data flows, but not necessarily what possible values each datum could hold, such as its type.

Additionally, an execution unit has an *apparent interface*, which formally specifies the data that flows into and out of it in a way that is verified by the programming language implementation. This information can appear in the uses (invocations) of the execution unit, or its declaration, or both. The apparent interface is the information the programmer has available as they code to understand how information flows through the program.

We say that an execution unit has *interface integrity* if its effective interface is equal to its apparent interface. A programming language with interface integrity is one in which every execution unit written in the language has interface integrity.

 doi:10.20944/preprints202509.0517.v1

Interface integrity, therefore, demands that programs not use global or static variables, or perform input/output operations, without explicitly specifying this in the apparent interface of all execution units that can modify or be influenced by globals. Prohibiting global variables prevents *common coupling*, one of the highest levels of coupling identified by Stevens *et al.* [2]. It also forbids destructive modification of data structures, unless the apparent interface of the execution unit doing so documents specifically what may change.

It does not, however, place any restrictions on a language's type system. It also does not preclude reassigning local variables, as long as the change cannot be observed outside the execution unit. Therefore, it does not preclude imperative or object-oriented programming.

Pure functional programming languages, such as Haskell and Clean, naturally exhibit interface integrity, as do pure logic programming languages, such as Mercury. Rust comes close to exhibiting interface integrity, except that it does not require that function signatures document the input/output they perform. Note that, because of Rust's ownership model, Rust programs can mutate data structures without violating interface integrity, because the mutation is documented in the apparent interface, and ownership ensures that the mutation cannot be observed other than through the apparent interface.

One important benefit of interface integrity is the absence of "spooky action at a distance" — values of variables local to an execution unit do not change unless it explicitly changes them. Because the apparent interface is checked by the compiler, the programmer can be confident that they are aware of all data flow into and out of an execution unit. Another benefit is that the behaviour of an execution unit depends only on its inputs as specified in its apparent interface, and never on the history of the computation. These properties are important to ensuring separation of concerns, as it allows both for the programmer and user of an execution unit to consider the other only as far as dictated by the apparent interface.

We argue that interface integrity is the key characteristic of declarative programming languages that leads to their high productivity and robustness [8]. It does not shackle the programmer by preventing them from using imperative programming constructs in the components they develop, but does shield them from the effects of whatever the programmers of other components choose to do.

3. WYBE

In this section, we will introduce the fundamental concepts of the WYBE language. This is not intended to be a complete tour of the language, but rather to illustrate how the principles presented in Sections 1 and 2 can inform a programming language design. WYBE borrows from functional, logic, procedural, and, to a lesser extent, object-oriented programming languages. We begin by outlining the functional aspects.

3.1. Functions and Expressions

Interface integrity necessitates that WYBE functions are genuine functions: they cannot have side effects (since a function execution cannot affect anything outside itself), and always produce the same output given the same input (since they do not have access to any information other than their inputs and the information they compute from them).

The WYBE syntax for declaring and defining functions is fairly conventional, beginning with the **def** keyword, followed by the function signature, an **=** sign, and an expression. In this example, all types are inferred:

```
def fahrenheit_to_celsius(f) = (f - 32.0) / 1.8
```

Parameter and result types can be declared explicitly by following parameter names and the entire signature with a colon and the type, as in:

```
def fahrenheit_to_celsius(f:float):float = (f - 32.0) / 1.8
```

 doi:10.20944/preprints202509.0517.v1

Functions may have any number of inputs, and one output. The body of the function definition, following the "=", can be any expression, which, in addition to function calls, can include **let**, **where**, **if**, and **case** expressions. For example:

```
def absolute(i:int):int = if { i < 0 :: -i | else :: i }
```

Note the braces surrounding the body of the conditional, the vertical bar separating the cases, and the double colon separating each condition from the consequent expression. Any number of *condition*::*value* pairs may included, resulting in the *value* from the first succeeding *condition*.

The form of the **let** and **where** expressions are: **let** {*statements*} **in** *expression* and *expression* **where** {*statements*}. These are equivalent; in both cases, the sequence of statements is executed, presumably assigning some variables, and then the given *expression* is evaluated with those variable bindings in place. Statements will be discussed in the next two sections.

3.2. Procedures, Relations, and Statements

Procedures are generalisations of functions, implementing relations, thus they manifest the influence of both the imperative and logic programming paradigms. They can have any numbers of inputs and outputs, which may be intermixed. In fact, every function *is* a procedure whose last argument is an output, which is elided in the function declaration syntax.

Procedure declarations and definitions are similar to functions, beginning with **def**, followed by the procedure signature, and the body, which is a sequence of statements surrounded by braces. A procedure to relate Fahrenheit and Celsius temperatures could also be defined as:

```
def fahrenheit_celsius(f:float, ?c:float) {
    ?c = (f - 32.0) / 1.8
}
```

Note the question mark operator before each occurrence of the variable c. In the signature, the question mark indicates that c is an output parameter; in the definition body, it indicates that that variable is assigned, rather than used, by that statement. The value returned for each output parameter is the final value of that variable after execution of the procedure body. Only the presence of the question mark determines that a variable is to be assigned (or reassigned) by a statement; whether it appears on the left or right of an = is immaterial. The = statement is not an assignment, it is a symmetric equality constraint that simply makes its two arguments equal, if possible (see Section 3.4 for what happens if it is not possible). If a variable prefixed with a question mark already has a value, the old value is replaced. Variables need not be explicitly declared; their scope extends from their (first) assignment to the end of the procedure body. Any procedure call may assign as many variables as it has output parameters.

Allowing variable reassignment is one key difference between languages with interface integrity and declarative languages. WYBE makes most of the common imperative programming language constructs available, giving some useful flexibility to the programmer. Yet, because of interface integrity, the consequences of allowing assignment are completely restricted to procedures that use it. We argue that managing changing state within a single procedure is well within the mental capacity of a programmer, while managing changing state globally in a program is frequently unmanageable.

In keeping with the principle of irrelevance insensitivity, whether defined as a function or procedure, procedures can be called as an expression or statement. The following four statements are all equivalent:

```
    ?c = fahrenheit_to_celsius(f)
    ?c = fahrenheit_celsius(f)
    fahrenheit_to_celsius(f, ?c)
    fahrenheit_celsius(f, ?c)
```

 doi:10.20944/preprints202509.0517.v1

The pattern of input and output parameters of a procedure are is called its *mode*. WYBE permits multiple modes to be defined for a procedure, so we could also define:

```
def fahrenheit_celsius(?f:float, c:float) {
    ?f = c * 1.8 + 32.0
}
```

This makes `fahrenheit_celsius` a *bijection*, allowing it to convert between Fahrenheit and Celsius in either direction, using either functional or procedural (relational) notation. Both of these statements will convert from Celsius to Fahrenheit:

```
fahrenheit_celsius(?f, 37.0)
fahrenheit_celsius(?f) = 37.0
```

This ability to define both directions of a bijective function, and to use the different modes to run functions both "forwards" and "backwards", forms the basis for pattern matching in WYBE, as discussed in Section 3.6.

3.3. Imperative Programming

WYBE supports imperative programming through statement sequences and variable reassignment, as well as conditional statements and bounded and unbounded iteration. Conditional statements have the same form as conditional expressions, except that the consequents are statement sequences rather than expressions. For example:

```
def absolute(i:int, ?abs:int) {
    ?abs = i
    if { i < 0 :: ?abs = -abs }
}
```

Bounded loops are specified with the **for**...**in** statement, which iterates over sequences of values, such as lists or ranges. For example, these statements sum the first ten integers:

```
?sum = 0
for ?i in 1..10 {
    ?sum = sum + i
}
```

Unbounded loops are specified with the **do** {...} statement. Any sequence of statements can be written within the braces, including the special **while** and **until** statements, which specify when and under what conditions the loop should be terminated. The loop can also be terminated early with a **break** statement. For example, this sequence finds the first counting number that is divisible by 3, 4, and 5:

```
?n = 1
do{ until (n%3 = 0 & n%4 = 0 & n%5 = 0)
    ?n = n + 1
}
```

Of course, any code that can be written using iteration can also be written using recursion. However, iteration, variable reassignment, and iteration provide flexibility for the programmer, allowing them to choose the most convenient style for each task. They do not compromise interface integrity, because they have no impact outside the execution unit, and therefore code can be rewritten between imperative and declarative styles without affecting other execution units. This also ensures that the principle of irrelevance insensitivity is respected.

 doi:10.20944/preprints202509.0517.v1

3.4. Partial Functions, Partial Procedures, and Disjunction

WYBE explicitly supports *partial* functions and procedures, which may be unable to produce a result for some inputs. For example, the `head` function, which returns the first element of a list, is partial, because the empty list has no first element. A call to a partial function or procedure is said to be a partial expression or statement, meaning that it will *fail* when it cannot produce a result. It also renders the expression or statement sequence containing it partial.

Any function or procedure whose definition is partial must be explicitly declared to be partial by following the **def** keyword with **{ partial }**. Without this annotation, the definition is taken to be *total* (certain not to fail), and the compiler will report an error if it cannot prove the definition to be total.

Partial expressions are akin to option or Maybe types, in that the compiler ensures that a partial expression or statement has succeeded before allowing its value to be used. However, unlike option or Maybe types, the values produced can be used without explicitly checking them for success or "unwrapping" them. Any failure anywhere in a statement sequence or expression immediately causes the entire statement sequence to fail, reverting all assigned variables to the state they had the beginning of the statement sequence (whether undefined or holding a previous value). The particular case of a call to a partial procedure with no outputs is equivalent to a Boolean value, and can be used to impose constraints that must be satisfied for a statement sequence to succeed. In keeping with the principle of irrelevance insensitivity, a call to a partial procedure with no outputs can be treated as a Boolean expression, and vice versa.

Disjunction, specified with the vertical bar (`|`) operator, allows the programmer to specify alternative statements to be executed in case of failure. For example,

```
{ ?t = tail(lst) | ?t = [] }
```

will assign the tail of `lst` to `t`, but if the list is empty, it will instead assign `[]`. Note that multiple statements can be used in each disjunct, with braces used to delimit the scope of the disjunction operator. If any expression appearing in a statement fails, the whole statement fails, and if any statement in a disjunct fails, the disjunct fails and control transfers to the next disjunct. Calls to partial and total functions and procedures can be freely intermixed. Semantically, a sequence of statements is implicitly a conjunction.

Disjunction also works at the level of expressions, where each disjunct provides a possible value for that expression, with the first expression to succeed giving the value. For example, this is a more concise equivalent of the above example:

```
?t = (tail(lst) | [] )
```

In both of these examples, the second disjunct is total, so the disjunction is total. At the expression level, disjunction operates as a generalisation of the usual Boolean disjunction, where each disjunct carries a value in addition to its success or failure, with the value of the disjunction being the value associated with the first disjunct to succeed.

Another way to handle failure is to place a partial statement or expression in a condition of an **if** statement or expression, discussed in Sections 3.1 and 3.3. That is, conditions can involve calls to partial procedures or functions, with failure of any partial call causing the condition to fail.

3.5. Modules, Types, and Constructors

In line with the principle of modularity, every WYBE source file is considered to be a module whose name is taken from the file name, and in accord with the principle of information hiding, by default, all definitions in a module are private to that module. Definitions can be made public by prefixing them with the **pub** keyword.

To support data encapsulation, every WYBE type is a module, which can define the public operations on this type, as well as private operations useful in implementing the public ones. Thus,

 doi:10.20944/preprints202509.0517.v1

WYBE types are abstract data types, adopting one of the most useful aspects of object oriented programming.

A module becomes a type simply by declaring all its constructors separated by vertical bars (`|`), following the **constructors** keyword. A type can have any number of constructors. Constructors, unlike their counterpart in object-oriented languages, have no body; they simply produce a value that holds their arguments together with their constructor. They combine the features of structure or record declarations, enumerated types, and union or variant record types, from procedural languages. That is, they are *algebraic types* [9], providing a precise and concise data abstraction. Because they are also immutable, WYBE has *value semantics*, meaning two values constructed with the same constructor and arguments are interchangeable. Furthermore, strong typing ensures enforced consistency.

For example, the `bool` type is defined by the following in a file named `bool.wybe`:

```
pub constructors false | true
```

A `coordinate` type can be defined in a file named `coordinate.wybe`:

```
pub constructors cartesian(x:float, y:float)
```

A polymorphic type is defined by following the **constructors** keyword with type variables within parentheses, and then using those type variables as types in the constructor definitions. Type variables consist of a single capital letter followed by zero or more digits. For example, the `list` type is defined as:

```
pub constructors (T) [] | [head:T | tail:_(T)]
```

This declares a parametric type `list(T)` with two constructors, `[]` (the empty list) and `[`*h*`|`*t*`]` (a cons cell with head *h* of type `T` and tail *t* of type `list(T)`). This type uses the special list notation taken from Prolog, with elements within square brackets and separated by commas, with the final tail of the list separated by a vertical bar. The special type constructor `_` is always an alias for the type of the current module, avoiding the need for copious repetition of the current type name throughout all the constructors, functions, and procedures in the module.

For example, after importing the above types, the following will construct a `bool`, `coordinate`, and `list(int)` value (the last could be equivalently written "`[2]`"):

```
?b = true
?coord = cartesian(1.0, 2.0)
?lst = [2 | []]
```

3.6. Pattern Matching

The declared constructor functions can also be used backward, as deconstructors; for example, this code will deconstruct the above coordinate into variables `x` and `y`:

```
coord = cartesian(?x, ?y)
```

For types with more than one constructor, the deconstructors are partial; for example

```
[?hd | ?tl] = lst
```

will deconstruct a list (remember: `=` is symmetric), but can only be used in a context in which a partial statement is permitted, such as in a condition. This allows data to be deconstructed anywhere in the code, not just in special language constructs, as long as the program makes provision for the

 doi:10.20944/preprints202509.0517.v1

deconstruction to fail, if that is a possibility. For example, a partial function to find the n^{th} element (1 origin) of a list can be defined:

```
def {partial} nth(n:int, lst:list(T)):T =
    let { lst = [?hd | ?tl ] } in
    ( if { n = 1 :: hd
         | n > 1 :: nth(n-1, tl)
      }
    )
```

Here the pattern matching statement `lst = [?hd|?tl]` is permitted, even though it will fail when `lst` is empty, because the function in which it is used is declared **partial**. Likewise, the conditional expression has no **else** branch; this is permitted for the same reason. A call to `nth` with a non-positive element number will fail.

In addition, *accessor* and *mutator* functions are automatically generated for the named fields of the constructors. In the case of the `list(T)` type, these were specified as `head` and `tail`, which identify the first element of a list and the rest of the list, respectively. As with the list deconstructors, accessors and mutators are partial for types with more than one constructor. Mutators will be discussed in Section 3.8.

Although constructors, deconstructors, accessors, and mutators are automatically generated from **constructors** declarations, they are not otherwise special. A type declaration can be changed at will, substituting user-written functions for constructors, deconstructors, accessors and mutators, or vice-versa, without affecting any code that uses that type. For example, some of the constructors of a type could be replaced with functions that call other constructors, or the entire type can be redefined, as long as functions compatible with the original constructors, deconstructors, accessors and mutators can be written. This is in keeping with the uniform reference principle [6], and more generally the principle of irrelevance insensitivity, and means that the common advice for object oriented languages, to make fields (data members) of a class private, does not apply to WYBE, because the programmer may change the implementation at will while preserving the interface.

3.7. Variable Update

In addition to *using* the value of a variable (giving the variable name with no prefix operator) and *assigning* a variable (by prefixing the variable name with the **?** operator), WYBE also supports *updating* a variable (by prefixing it with the **!** operator). In the context of a parameter declaration, no prefix operator means call by value, a prefix **?** operator means call by result, and a prefix **!** operator means call by value-result. "In the latter case, the variable value is passed as both an input and an output of a call. For example, this can be used to define a generic procedure to exchange the values of two variables:

```
def swap(!x:T, !y:T) {
    ?t = x
    ?x = y
    ?y = t
}
```

which could then be used to put two variables in ascending order:

```
    if { smaller > larger :: swap(!smaller, !larger) }
```

This cannot violate interface integrity, since only local variables can be affected.

In recognition of the fact that imperative programs often apply a function to a variable, storing the result back into the same variable, WYBE allows one argument to a function to be a variable prefixed with the **!** operator, indicating that the function value should be stored back in that argument. This function call is then used as a statement, for its effect. This provides a convenient, more general

 doi:10.20944/preprints202509.0517.v1

replacement for built-in operators like `+=` and `*=` in languages of the C family. For example, the following statements increment a variable `p`, take the reciprocal of a variable `q`, and impose a lower bound of 0 on a variable `r`:

```
!p + 1
1.0 / !q
max(!r, 0)
```

3.8. Structure Mutation

This brings us to the question of mutation of data structures, commonly permitted in imperative languages, but not in declarative ones. Mutating a data structure would constitute data flow out of the execution unit. If the data structure is considered to be both an input and an output of the operation, however, it could be permitted, with an important caveat: only the arguments passed as both input and output should experience any mutation. If any other variables contain (part of) the data structure, they must not be changed.

One common way of addressing this is through the imposition of linear typing [10], which ensures that values are used exactly once in a function execution. This restriction prevents a value from having multiple references, or *aliases*, ensuring that mutating that value cannot affect any value other than the intended one. Linear typing, however, does impose some inconvenient restrictions, which are often not needed. The requirement to use each value requires unneeded values to be explicitly disposed of or passed through to the end of the computation. Hofmann [11] proposes an affine type system, which permits values to be used *at most* once, eliminating this issue. Nevertheless, data structures that will not be mutated do not need the restrictions of linear or affine types, forcing the programmer either to be constantly aware of which structures may be modified and which will not, or to treat all values as if they may be modified.

WYBE takes a different approach, providing non-destructive mutation operations, which return a fresh data structure that differs from the original only in that the specified field holds the specified new value. As discussed in Section 4, when the compiler can prove that the original data structure is not used after the update, this is optimised to instead destructively update that structure. This stands in contrast to languages with linear or affine types, or languages with an ownership model, such as Rust, which place the burden of deciding whether a data structure should be mutated or copied on the programmer. WYBE follows the principle of irrelevance insensitivity, only asking the programmer to determine what data structures should be generated, without needing to specify how they should be produced. This has the additional benefit of allowing the compiler to generate different code for the same operation depending on context, as discussed in Section 4. The trade-off is that the programmer must rely on the compiler to generate efficient code.

As discussed in Section 3.5, to support an imperative style of programming, WYBE automatically generates a mutator function for each member or field of a type. Also, as a syntactic convenience, the left-associative caret (`^`) operator implements right-to-left function and procedure application, so that `a^g^f` is equivalent to `f(g(a))`. This is meant to be reminiscent of the `a.g.f` syntax denoting field access in many imperative languages, but applies any function or procedure to an argument.

For example, the following statements create a coordinate, access its `x` member, update the coordinate by incrementing its `y` member, and finally exchange the coordinate's `x` and `y` members. After all these statements, the value of `x1` will be 3.0 and the value of `coord` will be `cartesian(5.0,3.0)`.

```
?coord = cartesian(3.0, 4.0)
?x1 = coord^x
!coord^y + 1.0
swap(!coord^x, !coord^y)
```

 doi:10.20944/preprints202509.0517.v1

It is important to note that, due to the non-destructive semantics of structure mutation, structure sharing (aliasing) is semantically irrelevant. A mutation operation never affects any variables that it does not explicitly mention. So, for example, although the second line below makes `coord` and `coord2` aliases, these statements would wind up with `coord` having the value `cartesian(4.0,3.0)`, while `coord2` would have the value `cartesian(3.0, 4.0)`:

```
?coord = cartesian(3.0, 4.0)
?coord2 = coord
swap(!coord^x, !coord^y)
```

Programmers do not need to be concerned about aliasing of data structures.

3.9. Resources

Like declarative languages, languages with interface integrity lack global (or static, or class) variables. These variables provide a mechanism for hidden communication between calls, making the behaviour of a call depend not just upon that call, but upon the entire history of the computation, leading to potentially mysterious behaviour. They do have some redeeming qualities, though: they reduce the need to explicitly pass values around, and ensure that those values are consistently named throughout the program.

WYBE provides *resources* to serve this purpose without violating interface integrity. A resource is like a parameter that appears in procedure declarations, but not in calls. Calls to procedures that use resources must be prefixed with an exclamation point operator (`!`), to ensure the programmer knows to be cautious about reordering this statement relative to others, and to consult the procedure declaration to see the full scope of its effects.

A resource is declared with the **resource** keyword, followed by the name and type of the resource. A procedure that uses this resource can be declared by adding a **use** clause to its declaration, specifying what resources it uses and whether each flows into the procedure (no prefix) or out (`?` prefix) or both (`!` prefix). For example, the following declares a resource `parsing` holding a string to be parsed. The `start_parse` and `end_parse` procedures set up the `parsing` resource, and test that the full string has been parsed, respectively. The `char` procedure removes and returns the first character from the `parsing` resource, failing if it is empty, while `space` uses `char` to consume the next character, checking that it is a space character. It takes advantage of another WYBE feature: a procedure call may supply an input argument where an output is expected, in which case the output value is compared to the specified value, failing if they are not equal. The `digit` procedure removes the first character if it is a digit, returning its numeric value. The `integer` procedure removes one or more digit characters, computing the integer value. This is style of parser is akin to a deterministic definite clause grammar [12], common in Prolog implementations.

 doi:10.20944/preprints202509.0517.v1


```
resource parsing:string

def start_parse(str:string) use ?parsing { ?parsing = str }

def {partial} end_parse use parsing { parsing = "" }

def {partial} char(?ch:char) use !parsing {
    [?ch | ?parsing] = parsing
}

def {partial} space use !parsing { !char(' ') }

def {partial} digit(?d:int) use !parsing {
    !char(?ch) & is_digit(ch)
    ?d = ord(ch) - ord('0')
}

def {partial} integer(?i:int) use !parsing {
    !digit(?i)
    do {
        while (!digit(?d))
        ?i = i * 10 + d
    }
}
```

The **use**...**in** statement creates a scope in which a resource is defined, permitting it to be assigned and used. This example statement uses the `integer` procedure to parse the string `"345 6789"` into the variables `n1` and `n2`, respectively. This statement sequence is partial, failing if any part of the sequence fails.

```
    use parsing in {
        !start_parse("345 6789")
        !integer(?n1)
        !space
        !integer(?n2)
        !end_parse
    }
```

Because the compiler ensures that all the resources used by a procedure are in scope for every call to that procedure, reporting an error if not, the programmer can be confident that resources will not be used or modified during execution of any procedure that does not explicitly state that it uses it. Also, only statements, not expressions, can use resources (function call syntax is not permitted for procedures that use resources), ensuring that the execution of expressions is independent of any resources, and that the order of evaluation of expressions does not affect program behaviour.

One important use of resources is for managing input/output (I/O). The `!io` resource represents the state of the world outside the program. This resource is automatically in scope at the start of the main program, thus procedures that **use** it can be invoked from the main (top-level) program. The type of the `io` resource is declared to be `unique` (equivalent to affine). Because there is no way for the user to create an `io` value, and the `io` resource must be present at the end of the main program, it is effectively linear, and all user procedures that perform input/ouput, or call any procedures that do, must be declared to **use** `!io`. For example, the following procedure would print "Hello, world!", would be called as `!hello`, and could only be called from a procedure declared to **use** `!io`:

 doi:10.20944/preprints202509.0517.v1


```
def hello use !io {
    !println("Hello, world!")
}
```

Some functional languages provide *monads* [13] to address this and other issues, allowing stateful operations to be defined as a composition of state transformations. This is a more powerful feature than resources, because monads can compose state transformations in complex ways. However, resources have one advantage over monads: a procedure can use many resources, which all are passed as inputs and/or outputs of that procedure alongside one another. The procedure may freely call other procedures that use any subset of those resources. In contrast, a function that wishes to use multiple monads must instead use monad transformers, nesting one monad within another, and using more complex techniques to access nested monads.

3.10. Higher Order Programming

Statement sequences can be treated as values, allowing them to be stored in variables, or passed as inputs or outputs to procedures or functions, by enclosing them in braces. Inside such anonymous procedures, an at-sign (@) followed by an integer n stands for its n^{th} parameter to the anonymous procedure. The @s can also be written without the following integers, in which case they are numbered from left to right in the statement sequence. Anonymous procedures can then be called by writing them followed by their parameters in parentheses. Typically, this means using the name of a variable holding an anonymous procedure as if it were a procedure name.

Anonymous functions are defined by enclosing an expression in parentheses, preceded by an @. As with anonymous procedures, anonymous functions can specify parameters with an @ followed by the parameter's ordinal number, and the ordinal numbers can be omitted, in which case the parameters are numbered in the order they appear. Again, an anonymous function is equivalent to an anonymous procedure, but with one extra output parameter.

Partial application is supported for both functions and procedures: a function or procedure call with too few arguments is taken to be an anonymous procedure or function with as many @ arguments as necessary.

Closures may be created by using variables from the enclosing scope in an anonymous function or procedure. The value of the variable is taken at the point the closure is created; subsequent assignments of the variable have no effect. For example, this code prints 3:

```
?n = 2
?add_n = @(@ + n)
?n = 5
!println(add_n(1))
```

Anonymous procedures and closures that are annotated with the {resource} prefix can invoke procedures that use resources. The type of such a value indicates that it may use some resources, so it can only be passed as a higher order argument whose type has the {resource} prefix to indicate that it is permitted to use resources. For example, a procedure to map a resourceful procedure over a list is defined in the WYBE library:

```
pub def map(f:{resource}(T), xs:list(T)) {
    for ?x in xs { !f(x) }
}
```

Note that, because the `f` parameter is specified to allow resources, the call to `f` must be preceded with an !. This procedure may then be called, for example, to print all the elements of a list:

 doi:10.20944/preprints202509.0517.v1


```
!map(println, [2,3,5,7,11])
```

4. Implementation

The WYBE compiler is implemented in Haskell, compiling all features (including imperative ones) to the declarative LPVM intermediate representation [14] for analysis and optimisation, and finally converting to native code via the LLVM compiler backend [15]. LPVM is a single-assignment form based on Horn clauses, making analysis and transformation easier.

Memory management is provided using the Boehm–Demers–Weiser garbage collector [16,17]. Data structures are represented using a variation of the algebraic data type representation for C programs of Naish *et al.* [18]. This representation uses tags stored in the low-order bits of pointers to encode constructors, which was shown to offer good performance.

The compiler takes advantage of the language's interface integrity. A statement whose outputs are not used is removed by the compiler, and one whose inputs are identical to an earlier statement in the same procedure body will be replaced by assignments reusing the outputs of the earlier call. Many optimisations such as constant folding, constant propagation, and common subexpression elimination are performed, unaffected by intervening procedure calls. Calls to procedures with small bodies are unfolded, avoiding the cost of the procedure call, and allowing other optimisations to apply in more circumstances.

The compiler employs alias analysis and live variable analysis to determine when a copying mutation operation references a data structure that has no other references and will not be used again, allowing the compiler to transform the copy-and-mutate operation into a destructive mutate. A major limitation on this optimisation is that data structures passed into a procedure must be assumed to be aliased or live, because the caller may continue to use it. To make this and other optimisations more applicable, the compiler uses multiple specialisation [19,20] to create a variant version of such a procedure for cases where the argument is known not to be aliased or used later, allowing the optimisation to apply to mutations of data structures passed as arguments.

Compile-time garbage collection [7,21,22] is a related optimisation applied by the WYBE compiler. This applies when the compiler needs to allocate storage for a new data structure while an existing data structure of the same size is known not to be aliased and not to be used again. In this case, the memory of the existing data structure can be used instead of allocating fresh storage and needing to garbage collect the old structure. In some cases, some values to be written to the new structure already exist in the same place in the old one, saving the need to store them.

The compiler makes use of LLVM's last call optimisation to optimise procedure calls immediately followed by an operation to return to the caller into a direct jump to that procedure. This turns tail recursive procedures into loops. The WYBE compiler extends this to employ tail recursion modulo construction optimisation [23,24], which applies where the only operations between the recursive call and the return allocate memory and store into it. In this case, as many of the operations as possible between the final call and the return are moved before the call, leaving only instructions that write results of the final call into memory. Then the procedure is specialised so that instead of an output paramter to carry its output, it accepts an *input* that points to the memory into which the value is to be written. The call to the procedure is then modified to pass a pointer to the location to which the result was to be written, and the instruction to write it is removed. Once this is done, last call optimisation applies. The combination of this and the above optimisations allows the compiler, for example, to optimise a call to a recursive procedure to append two lists, in cases where the first list is known not to be aliased or used again, into a procedure that iterates down the first list replacing the final NULL pointer with a pointer to the second argument.

Some optimisations performed by the compiler require that the procedures called by a given procedure (its *callees*) be compiled and analysed before the procedure itself. To achieve this, the compiler compiles modules bottom-up; that is, where one module imports another, it compiles the

 doi:10.20944/preprints202509.0517.v1

other module before the one that imports it. Where modules import one another, they are compiled together. Likewise, each procedure is compiled after all its callees, with mutually recursive procedures compiled together. The compiler writes the analysis results for the procedures in a module to the object file of the module, so a module only needs to be compiled if it is changed or depends on a module that has changed.

Multiple specialisation requires the compiler to have information about the callers of a procedure to determine whether it should be specialised. Because modules and procedures are compiled bottom-up, this information is not available at the time each procedure is compiled. This is handled when the program executable file is being generated, by analysing modules and procedures in the reverse order: with all callers to a procedure analysed before the procedure itself. When compiling a procedure that fulfils the requirements for a more efficient specialised version, the compiler decides whether the benefit of generating a faster, specialised version outweighs the cost of increased code size.

The compiler is released as open source software under the terms of the MIT License. It can be downloaded from GitHub[2].

5. Conclusions

This paper has presented a set of programming language design principles aimed at supporting development of a wide spectrum of practical applications. These principles are organised around three key goals. Above all, we argue, a programming language should prioritise the easy development of robust programs. One key principle supporting this goal is interface integrity: the principle that each executable part of the program should have a formal interface that specifies the data that can flow into and out of it, and no data can flow other than through these interfaces. This restriction ensures that the developer and user of each part are aware of all the information that can influence the behaviour of that part, and all the ways that part can influence the behaviour of the rest of the program. It ensures that there can be no "spooky action at a distance", where data changes for no apparent reason.

The secondary goal of a language should be to facilitate development of code that can be maintained and evolved with minimal regression and minimal need for wide-ranging modification. We place program efficiency as the third priority; for some applications, it is paramount, but we argue that for most code, efficiency is less important than correctness and maintainability.

We have presented the WYBE programming language, which is being developed to explore these principles. WYBE combines features of functional, procedural (imperative), and logic programming languages, as well as borrowing from object-oriented languages. From functional (and, more generally, declarative) languages, it takes its value semantics, meaning that a value is what it is, and cannot be modified. It also borrows the expressiveness and precision of algebraic types. From imperative languages, it takes the ability to reassign variables, as well as statement sequences and looping constructs. Because of interface integrity, the effects of these constructs are limited to the scope of one execution of the procedure they appear in. We argue that this compromise gives the developer of an individual procedure flexibility to implement it as they wish without imposing any consequences on users of the procedure.

From logic programming languages, WYBE takes the idea that procedure calls can fail, undoing any assignments they have made. This capability allows pattern matching to be used at any point in the program, while ensuring that all possible forms of a data structure are handled.

To mitigate the problem of procedures with too many arguments that would tend to arise in languages with interface integrity, WYBE supports resources, which act as parameters that do not appear in argument lists, yet are automatically passed from one procedure to another. Because they appear in the declaration of every procedure that uses them, they do not compromise interface integrity.

Much work remains to be done to improve the language and implementation. One key feature WYBE lacks is bounded type quantification, such as type classes [25] in Haskell or interface types [26]

[2] https://github.com/pschachte/wybe

 doi:10.20944/preprints202509.0517.v1

or abstract classes [27] in Java. We would also like WYBE to support multithreaded execution. Currently, anonymous procedures cannot close over output arguments; in some cases this would be very expressive, and we would like to allow it. Adding support for more logic programming constructs, especially deep backtracking, would also be beneficial. In the domain of performance, many more optimisations are possible, including specialising higher order procedures for their function argument(s), improving memory management, and improving the pointer and liveness analysis to turn structure copy-and-update into destructive update in more cases.

Funding: This research received no external funding.

Data Availability Statement: Source code for the WYBE compiler is available at https://github.com/pschachte/wybe.

Acknowledgments: The author acknowledges and thanks the following for their many contributions to the implementation of the WYBE compiler: Ashutosh Rishi Ranjan, Ting Lu, Zijun (Zed) Chen, Rudolph Almeida, Marco Ho, James Barnes, Leyan (Terry) Lin, and Chris Chamberlain.

References

1. Banker, R.; Davis, G.; Slaughter, S. Software Development Practices, Software Complexity, and Software Maintenance Performance: a Field Study. *Management Science* **1998**, *44*, 433–450. https://doi.org/10.1287/MNSC.44.4.433.
2. Stevens, W.P.; Myers, G.J.; Constantine, L.L. Structured Design. *IBM Systems Journal* **1974**, *13*, 115–139.
3. Dijkstra, E.W. On the role of scientific thought.
4. Nygaard, K.; Dahl, O.J., The development of the SIMULA languages. In *History of Programming Languages*; Association for Computing Machinery: New York, NY, USA, 1978; p. 439–480.
5. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **1972**, *15*, 1053–1058. https://doi.org/10.1145/361598.361623.
6. Meyer, B. *Object-Oriented Software Construction*; Prentice Hall, 1988.
7. Giuca, M. Mars: An Imperative/Declarative Higher-Order Programming Language With Automatic Destructive Update. PhD thesis, The University of Melbourne, 2014.
8. Hudak, P.; Jones, M.P. Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. *Contract* **1994**, *14*, 0153.
9. Burstall, R.M.; MacQueen, D.B.; Sannella, D.T. HOPE: An experimental applicative language. In Proceedings of the Proceedings of the 1980 ACM Conference on LISP and Functional Programming, New York, NY, USA, 1980; LFP '80, p. 136–143. https://doi.org/10.1145/800087.802799.
10. Wadler, P. Linear types can change the world! In Proceedings of the Programming concepts and methods. Citeseer, 1990, Vol. 3, p. 5. Issue: 4.
11. Hofmann, M. A Type System for Bounded Space and Functional In-Place Update—Extended Abstract. In *Programming Languages and Systems*; Goos, G.; Hartmanis, J.; Van Leeuwen, J.; Smolka, G., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2000; Vol. 1782, pp. 165–179. Series Title: Lecture Notes in Computer Science, https://doi.org/10.1007/3-540-46425-5_11.
12. Pereira, F.C.N.; Warren, D.H.D. Definite Clause Grammars for Language Analyisis–A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* **1980**, *13*, 231–278.
13. Wadler, P. How to Declare an Imperative. *ACM Computing Surveys* **1997**, *29*, 240–263.
14. Gange, G.; Navas, J.A.; Schachte, P.; Søndergaard, H.; Stuckey, P.J. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming* **2015**, *15*, 526–542.
15. Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, 2004.
16. Boehm, H.J.; Weiser, M. Garbage collection in an uncooperative environment. *Software: Practice and Experience* **1988**, *18*, 807–820. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380180902, https://doi.org/10.1002/spe.4380180902.

 doi:10.20944/preprints202509.0517.v1

17. Boehm, H.J.; Demers, A.J.; Shenker, S. Mostly parallel garbage collection. In Proceedings of the Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, New York, NY, USA, 1991; PLDI '91, pp. 157–164. https://doi.org/10.1145/113445.113459.
18. Naish, L.; Schachte, P.; MacNally, A.M. Adtpp: lightweight efficient safe polymorphic algebraic data types for C. *Software: Practice and Experience* **2016**, *46*, 1685–1703. https://doi.org/10.1002/spe.2407.
19. Puebla, G.; Hermenegildo, M. Implementation of multiple specialization in logic programs. In Proceedings of the Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '95, La Jolla, California, United States, 1995; pp. 77–87. https://doi.org/10.1145/215465.215561.
20. Winsborough, W. Multiple specialization using minimal-function graph semantics. *The Journal of Logic Programming* **1992**, *13*, 259–290. Publisher: Elsevier.
21. Hughes, S. Compile-Time Garbage Collection for Higher-Order Functional Languages. *Journal of Logic and Computation* **1992**, *2*, 483–509. https://doi.org/10.1093/logcom/2.4.483.
22. Mazur, N. Compile-Time Garbage Collection for the Declarative Language Mercury. PhD Thesis, Katholieke Universiteit Leuven, 2004.
23. Friedman, D.P. *Unwinding structured recursions into iterations* /; Indiana University, Computer Science Department,: Bloomington :, 1974.
24. Warren, D.H.D. An improved Prolog implementation which optimises tail recursion. DAI research paper 141, Dept of Artificial Intelligence, University of Edinburgh, 1980.
25. Wadler, P.; Blott, S. How to make ad-hoc polymorphism less ad hoc. In Proceedings of the Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, 1989; POPL '89, pp. 60–76. https://doi.org/10.1145/75277.75283.
26. Agesen, O.; Freund, S.N.; Mitchell, J.C. Adding type parameterization to the Java language. *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* **1997**. https://doi.org/10.1145/263698.263720.
27. Gosling, J.; McGilton, H. *The Java Language Environment: A White Paper*; 1995.