are procedures procedures3.45

The Wybe Programming Language

Peter Schachte

January 26, 2018

The aim of the Wybe programming language is to be simple, easy-to-learn, and easy-to-use, while supporting large teams in producing maintainable efficient programs. It encourages designs with low coupling and high cohesion, and takes the best aspects of both declarative and imperative programming, in particular taking ideas from procedural, object-oriented, functional, and logic programming languages.

1 Lexical structure

Comments begin with a hash (#) character and continue until the end of the line.

Identifiers are sequences of letters, digits, and underscore characters not beginning with a digit. Numbers follow C's syntax, except that they may contain (but not begin with) underscore characters (_), which are ignored, and can be used to make large numbers easier for humans to read. Operator symbols consist of any sequence of one or more of the following characters:

```
~ @ $ % ^ & + - * / = | \ : ; < > .
```

Any sequence of any of these characters plus any parenthesis, bracket, or brace characters, written between backquote characters (`) is an ordinary identifier *(not yet implemented)*. That is, surrounding an operator with backquotes allows it to be used as an identifier.

String literals follow orthodox syntax: they begin and end with a double quote character ("). Inside the string, the following sequences have the specified special meaning:

sequence	**meaning**
\a	alert/bell (ascii code 0x07)
\b	backspace (ascii code 0x08)
\e	escape (ascii code 0x1B)
\f	formfeed (ascii code 0x0C)
\n	newline (ascii code 0x0A)
\r	carriage return (ascii code 0x0D)
\t	horizontal tab (ascii code 0x09)
\u*hhhh*	the unicode character with code 0x*hhhh* (where *h* is a hex digit) *(not yet implemented)*
\U*hhhhhhhh*	the unicode character 0x*hhhhhhhh* (where *h* is a hex digit) *(not yet implemented)*
\v	vertical tab (ascii code 0x0B)
\x*hh*	the character 0x*hh* (where *h* is a hex digit)
\(*expr*)	the value of expression *expr* as a string (for any expression *expr*)
c	the character *c* (for any other character *c*)

To include a double quote or backslash (\) character in a string, precede it with a backslash.

Character constants begin and end with a single quote, but can only contain a single character. All the character escapes for strings can be used in character constants, except for \(*expr*).

2 Syntax

The primary syntactic building blocks of Wybe programs are statements and expressions. In this section, we will discuss their syntax; their meaning and the relationship between them will be discussed in Section 3.

Both statements and expressions have the same usual syntax:

$$name(expr_1,\ldots,expr_n),$$

where *name* is an identifier naming the procedure called by the statement or the function called by the expression, and there are any number of expression arguments, separated by commas. If there are no enclosed expressions (i.e., $n = 0$), then the empty parentheses are omitted. Note that the open parentheses, if present, must immediately follow the name, with no intervening whitespace.

To make code more readable and convenient, Wybe provides several special syntactic forms. All of these forms are simply syntactic sugar, and can always be written as the equivalent code in the usual syntax shown above.

First, operator symbols can be used as infix or prefix operators (there are no postfix operators, aside from function call and indexing). It is not necessary to explicitly declare operators; any operator symbols can be used by the programmer. Operator precedence and associativity are determined by the fixity (prefix or infix) and the final character of the operator, as follows:

token	**fixity**	**associativity**	**precedence**
identifier, string, or number	non-operator	non-associative	16
bracketed expression	non-operator	non-associative	16
`.` *(a single period)*	infix	left	15
`(...)` *(application)*	postfix	left	15
`[...]` *(indexing)*	postfix	left	15
`?` *or* `!`	prefix	non-associative	14
`:` *(a single colon)*	infix	non-associative	13
any other operator	prefix	non-associative	12
operator ending with `^` *or* `@`	infix	non-associative	11
operator ending with `*` *or* `/` *or* `%`	infix	left	10
operator ending with `-` *or* `+`	infix	left	9
operator ending with `.`	infix	left	8
operator ending with `%` *or* `$`	infix	right	7
operator ending with `\` *or* `~`	infix	left	6
operator ending with `<` *or* `>`	infix	non-associative	5
operator ending with `&`	infix	left	4
operator ending with `|`	infix	left	3
operator ending with `=`	infix	right	2
operator ending with `:`	infix	right	1
declared distfix operator	distfix	non-associative	0
operator ending with `;`	infix	left	-1

Bracketed expressions, those surrounded by parentheses, square brackets, or braces, are treated differently depending on whether they are preceded by whitespace.

If the bracketed expression e is immediately preceded by an identifier, string, number, or bracketed expression d, with no intervening whitespace, then e is expected to be at precedence 0. That is, it is a comma-separated sequence of precedence 1 expressions, and is interpreted as an expression $b(d, e_1, \ldots, e_n)$, where b is either `` `(...)` ``, `` `[...]` ``, or `` `{...}` ``, depending on whether the expression is surrounded by parentheses, brackets, or braces, and the $e_1, \ldots, e_n$ are the expressions separated by commas.

If the bracketed expression *is* preceded by whitespace, then it is treated as an expression in its own right, rather than as a postfix operator. Following whitespace, a single expression at precedence 1 or above surrounded by parentheses are interpreted as the enclosed expression. This is

simply used for grouping. A sequence of expressions at precedence 1 or above surrounded by square brackets and separated by commas optionally ending with a vertical bar and a final expression, are interpreted as a list. For example, `[a,b,c|d]` is interpreted as `‘[|]‘(a,‘[|]‘(b,‘[|]‘(c,d)))`. If the `|d` part is omitted, it defaults to `[]`.

Within braces following whitespace, one or more expressions at precedence -1 or above separated by vertical bars, or one or more expressions at precedence 0 or above separated by commas, will be read a set. For example, `{a,b,c}` is interpreted as `‘{,}‘(c,‘{,}‘(b,‘{,}‘(a,{})))`. Sets with elements of the form a`::`x are handled specially. For example, `{a::x,b::y,c::z}` is interpreted as `‘{:,}‘(c,z,‘{:,}‘(b,y,‘{:,}‘(a,x,{})))`.

Distfix operators, *i.e.* interspersed operator keywords and expressions, must be declared before they can be used. The declaration syntax is as either of the following:

`distfix` $ident_1$ sep_1 ... $ident_n$

where each of $sep1$ through $sepn-1$ is either `.` meaning no intervening expression or `..` meaning an expression should come at that point (and e). For example,

```
distfix if .. then .. else ..
```

Possible improvement: Introduce “associative” operators, which collect all the arguments separated by multiple occurrences of the same operator into a single call. This would allow `a < b < c < d` to be meaningful, and would handle `a + b + c + d` as a single operation. It would also handle { `a :: b c ; d :: e f g ; h :: i` } nicely. But this requires some kind of varargs facility, which we don’t have yet.

3 Statements and expressions

A *statement* is a Wybe construct that has an effect when executed, and does not have a value. Effects may include performing input or output, or setting the value of a variable. The usual form of a statement is

procname`(`*expr*,...`)`

where the *procname*, naming the procedure called, is an identifier, and there are any number of expression arguments, separated by commas.

An *expression* is a construct that determines outputs from inputs. Each expression is either a *variable reference* (described below) or a call to a *function*, of the form

funcname`(`*expr*,...`)`

where *funcname* is the name of the function to be called.

3.1 Dataflow

Each variable reference may either read, write, or modify (*i.e.*, both read and write) the variable. This is the *mode* of the reference. Prefixing a variable name with a `?` operator specifies that the variable will be written (without being read); this is an *output* mode. Prefixing a variable name with a `!` operator signifies that it will be read and written, an *input/output* mode. Unadorned variable names indicate read-only usage, an *input* mode.

Expressions have modes, too. An expression all of whose arguments are inputs is an input mode, while one all of whose arguments are outputs is an output mode. If there is a mix of inputs and outputs, or if any argument is in input/output mode, the expression is in input/output mode. This will be discussed further when we address tests.

3.2 Mode overloading

The mode of a statement is the ordered list of the modes of the arguments of the statement. This is specified in the declaration of the procedure. A statement may have any number of inputs and any number of outputs.

There may be more than one procedure with a given name, as long as the modes are different. The mode of the procedure call is used to determine which procedure with that name is being called. For example, the procedure `=` has a mode with the first argument output and the second input, and another with the first input and the second output. Thus `x` can be assigned the value `42` with either of the following statements:

```
?x = 42
42 = ?x
```

3.3 Defining functions and procedures

Functions and procedures are defined using the following syntaxes:

`func` *name*`(`*param*`:`*ptype*, ...`):`*rtype* `=` *expr*
`proc` *name*`(`*mode param*`:`*ptype*, ...`)` *stmt* ... `end`

where *name* and *param* are identifiers, *ptype* and *rtype* are argument and result type specifications, respectively, and each *mode* is either `?`, `!`, or nothing, meaning that that argument is output, input/output, or input only, respectively. Any number of parameters can be specified; if no parameters are specified, the empty parentheses are omitted. Any `:`*ptype*s or `:`*rtype* can be omitted, in which case the compiler will infer it, provided it is uniquely determined and the procedure or function is not exported (see Section 4.1). However, it is considered to be good programming style to include the declarations.

3.4 Functions *are* procedures

A function of n arguments is just a procedure with n input arguments plus a single output argument. A statement one of whose arguments is a function call is equivalent to two (or three) statements. For example, the statement

```
p(f(x,y))
```

is equivalent to

```
f(x,y,?temp)
p(temp).
```

In fact, this is how Wybe implements such a statement. If the expression is in output mode, the statement for the expression comes after; for example

```
p(f(?x,?y))
```

is equivalent to

```
p(?temp)
f(?x,?y,temp).
```

Where an expression is in input/output mode, it is equivalent to two procedure calls; for example

```
p(f(!x,!y))
```

is equivalent to

```
f(x,y,?temp)
p(!temp)
f(?x,?y,temp).
```

Conversely, any procedure can be called as a function by simply substituting a call to the procedure with its final argument omitted for the omitted argument. Thus

`func` *name*`(`*param*$_1$`:`*type*$_1$`,`...`,` *param*$_n$`:`*type*$_n$`):`*type* `=` *expr*

is exactly equivalent to

`proc` *name*`(`*param*$_1$`:`*type*$_1$`,`...`,` *param*$_n$`:`*type*$_n$`,` `?temp:`*type*`)` `?temp =` *expr* `end`

For example, the following is valid:

```
proc succ(x:int, ?y:int) ?y = x + 1 end
proc succ(?x:int, y:int) ?x = y - 1 end

proc updown(in, ?out) # out always = in
    ?tmp = succ(in)   # increment in
    tmp = succ(?out)  # decrement tmp
end
```

3.5 Tests

A *test* is a statement that can succeed or fail. A test is similar to a Boolean expression, except that none of the outputs of a test are deemed to have been produced if the test fails; if it succeeds, all the outputs are taken to have been produced. A statement sequence fails if any of its statements fails, and execution does not continue following the failing statement.

The condition of a conditional statement must be a test, as must the parts of a conjunction, disjunction, or negation. A conjunction, disjunction, or negation is a test, as is a call to a test procedure.

Outputs of a test may only be used in contexts where the test must have succeeded; in particular in the *then* branch of a conditional or following conjuncts in a conjunction. Outputs of a negated test can only be used when the negation fails. Variables that are assigned by all disjuncts in a disjunction, or all branches of a conditional may also be used, as may variables that are assigned prior to the conjunction, disjunction, negation, or conditional. As a convenience, a Boolean expression may be used as a test.

Note that some modes for a procedure name may be tests while others are not. A function may also be a test, which indicates a partial function. A statement is a test iff the called procedure is a test in that mode, or if any of its arguments is a test. An expression is a test iff the function is a test in that mode, or if any subexpression is a test. A test statement succeeds iff all argument expressions succeed *and* the procedure call itself succeeds. An expression succeeds iff all subexpressions succeed and the function call itself succeeds. Non-test procedure and function calls are treated as succeeding, and nested expressions are treated as the conjunction of their subexpressions. Thus a disjunction of statements is deemed to succeed if all the subexpressions of at least one of them succeeds.

Unlike in logic programming, tests in Wybe are deterministic: the output bindings of the first successful test are final. However, Wybe also supports generators *(not yet implemented)*, which are used in loops and are similar to backtracking in logic programs.

3.6 Implied modes

It is always permitted to supply an input in a mode where an output is expected, providing the type of that argument permits equality testing. In this case, the output mode is used to store the result in a temporary variable, and then the supplied value is tested for equality with the temporary variable. As we will see, data constructors are functions that work in forwards and backwards modes, making these implied modes useful for pattern matching.

3.7 Selection and iteration statements

The new syntax philosophy is conceptual parsimony, built on relatively few syntactic elements.

Miscellaneous ideas:

- Special syntax for manifest constant maps:

  $$\{key_1 :: value_1,\ key_2 :: value_2,\ \dots\}$$

  As a special case, sets can be written as

  $$\{key_1,\ key_2,\ \dots\}$$

- Allow use of "`|`" as an alternative to "`,`" to separate set and map items. When values involve multiple statements, this is useful because `|` has lower precedence than statement separation. If "`|`" is the first or last thing within curly braces, it is ignored, to simplify code layout and avoid the separator vs. terminator problem.

- Capitalising on this, we use this syntax for case statements:

  `case` *expr map*

  where the values in the *map* are statement sequences. Case expressions are similar, but where the values in the *map* are expressions.

  One challenge to this is that we want case values to be patterns, which can have outputs. This makes maps quite powerful, as matching keys must involve pattern matching, *i.e.*, executing code. Is this too powerful?

- Conditional statements and expressions have the syntax

  `if` *map*

  which is equivalent to `case true` *map*. For this to be meaningful, the semantics of manifest constants maps must be that multiple mappings with identical keys are permitted, and the *first* mapping for the repeated key is taken.

- Allow a set of statement sequences to be treated as a closure? A singleton set is a deterministic statement sequence (providing its statements are det), and multi-element sets are nondet, so "`|`" could be used to separate alternatives in generators. This would allow curly brace notation for statement sequences, but int means that where a statement sequence is used, it needs to be surrounded by braces. Perhaps we could coerce a single statement to a singleton statement sequence.

- Either maintain the decision not to have any statement separators, or allow separator (semicolon, I assume) to be omitted when its the last thing on a line.

- As a special syntactic case, allow proc calls with a single argument to be written without the parentheses surrounding the argument (or with them).

At the highest (tightest) precedence, Wybe has manifest constants, identifiers, and calls. A call is written as an identifier followed by an open parenthesis, a comma-separated argument list, and a close parenthesis.

Abandon this: At the loosest precedence, Wybe supports "interfix" operators: alternating identifiers and terms of higher precedence. These are parsed as a term whose constructor is the concatenation of the identifiers, separated by underscores, with the subterms, in order, as arguments.

Definition

```
def foo(x:int, y:int) use io {
    !bar(x)
    !baz(y)
}

def hypot(x:int, y:int):int = sqrt(x**2 + y**2)
```

Selection

```
if {
| prime(x):
    !bar(x)
    !baz(y)
| else:
    !baz(y)
    !bar(x)
}

case x {
| 0:: !bar(x)
      !baz(y)
| 1:: !baz(y)
      !bar(x)
| _:: nop
}
```

Looping

```
do {stmt1
    while test
    stmt2
}
```

4 Modules

Each Wybe module should be written in a file named *module*`.wybe`, where *module* is the name of the module. Module names may be any valid identifier, except for the special name `_`. Modules may be nested to any depth, allowing them to fulfill the role packages play in other languages. Modules may contain any number of the following things, in any order:

- Module imports
- Type definitions
- (Sub-)modules
- Function definitions
- Procedure definitions
- Resource declarations
- Statements

A module may also be an operating system directory, in which case all modules in that directory are taken to be its public submodules. If the directory contains a Wybe source file named `_.wybe`, this overrides the public importation, allowing the module to chose which submodules to export, as well as to define its own top-level procedures, functions, and initialization statements.

4.1 Exports

Each type, submodule, function, procedure, and resource definition may be preceded with the modifier '`public`', in which case, that element is exported. Without a `public` modifier, it is private, and cannot be used outside that module.

4.2 Imports

Wybe supports the following varieties of module importation:

- Module members may be imported such that they can be used without module qualification (`import`), or so that they can only be used with explicit module qualification *(not yet implemented)* (`use`). (Currently, `use` means import so that it can be used with or without module qualification).
- All public members of a module may be imported (`use` or `import` *module*), or only certain specified members may be imported (`from` *module* `use` or `import` *items*).
- Members may be imported such that they become public members of the *importing* module, visible to any module that imports it (by preceding the `use` directive with `public`) or not.

This gives the following eight varieties of importation:

`use` *module*	Imports everything made public by *module* so that it can be used with module qualification.
`import` *module*	Imports everything made public by *module* so that it can be used with or without module qualification.
`from` *module* `use` *items*	Imports the specified items, which must all be made public by *module*, so that they can be used with module qualification.
`from` *module* `import` *items*	Imports the specified items, which must all be made public by *module*, so that they can be used with or without module qualification.
`public` `use` *module*	Imports everything made public by *module* so that it can be used with qualification, and reexport it from the importing module as if it were defined there.
`public` `import` *module*	Imports everything made public by *module* so that it can be used with or without module qualification, and reexport it from the importing module as if it were defined there.
`public` `from` *module* `use` *items*	Imports the specified items, which must all be made public by *module*, so that they can only be used with module qualification, and reexport them from the importing module as if they were defined there.
`public` `from` *module* `import` *items*	Imports the specified items, which must all be made public by *module*, so that they can be used with or without module qualification, and reexport them from the importing module as if they were defined there.

4.3 Submodules

Submodule declarations have the form:

`module` *modname* `is` *item* `...` `end`

where *modname* is the submodule name and the *item*s are the contents of the submodule. Submodules are implicitly imported into the parent module.

4.4 Top level statements

Statements written at the top level of a file are executed, in the order written, at the time the program begins execution. Top level statements of a module are executed before the top level statements of any module that imports it, but are only executed once regardless of how many modules import it, making them suitable for module initialisations. Top level statements also fill the role a `main` function or method fills in other languages.

5 Types

Wybe is strongly typed, with parametric polymorphism (generics), type inference, abstract types *(not yet implemented)*, and subtypes *(not yet implemented)* supporting the Liskov substitution principle.

Type declarations have the form:

`type` *typename*(*paramname*,...) `=` *ctorname*(*membername*`:membertype`,...) | ... `end`

where *typename*, *paramname*, *ctorname*, and *membername* are identifiers, and *membertype* is a type. Here *typename* is the name of the type being defined, and the *paramname*s name the type parameters. If no type parameters are specified, the empty parentheses are omitted. Each *ctorname* is a data constructor for this type, and the *membername*s and *membertype*s are the arguments of the constructor and their types. Again, if a *ctorname* has no parameters, the empty parentheses are omitted. As many constructors may be specified as desired, separated by vertical bars.

For example:

```
type colour = colour(red:int, green:int, blue:int) end

type list(t) =
        empty
      | cons(head:t, tail:list(t))
end
```

It should be noted that Wybe constructors are not like constructors in object oriented languages, in that they do not have explicit implementations. Constructors simply construct a value holding the supplied data. But Wybe constructors are ordinary Wybe functions, except that they are not explicitly defined. You are free to define other functions to construct values of that type, and their use will not be distinguishable from constructors.

Wybe also provides a backward mode for each constructor, to serve as a deconstructor. That is, constructors can be run backwards to deconstruct a data structure. Additionally, a single argument deconstructor function is defined for each constructor argument. For types with more than one constructor, these backward deconstructor modes, as well as forward deconstructor functions, are tests. For the examples above, Wybe will support to following statements (where `c` is a color and `l` is a list):

```
# to deconstruct a color into red, green, and blue components
c = colour(?r, ?g, ?b)

# the following are equivalent:
?r = red(c)
?g = green(c)
?b = blue(c)

# to deconstruct a non-empty list into head and tail:
if l = cons(?h, ?t) ...
```

```
# to pattern match a singleton list:
if l = cons(?h, empty)
```

The Wybe library supplies a useful little function with only a backward mode:

The language also supports Myer's uniform access principle. Every type is simply a module whose name may be used as a type name. The primitive operations (methods) of the type are the operations exported by the module.

Constructors (and deconstructors) are ordinary functions. Each type declaration includes all its data constructors, each of which looks like a function declaration, except that the return type and function body are omitted. The compiler automatically generates the implementation of the constructor and deconstructor functions. However, constructors and deconstructors are ordinary functions, so an existing type with defined constructors and deconstructors can be redefined with new functions implementing the same interface as the previous constructors and deconstructors.

Functions and procedures can be declared to be `abstract`, in which case their implementations are omitted. The compiler will not permit abstract operations to be invoked; the value must be known to be of a subtype that implements the operation for it to be used.

There is an abstract type `data` that serves as the root of the hierarchy of data types. This type has several useful abstract operations such as equality testing, and printing. All types declared with constructors, as well as all the primitive types such as integer, float, and char, implement this type.

6 Functions and procedures

Wybe supports both functions and procedures. The top level of a procedure definition, and the top level of a module, are statement contexts. Calls written there are taken to be procedure calls. The arguments of a procedure or function call, and the body of a function definition, are expressions. Calls written as parts of expressions are taken to be function calls.

However, procedures and functions are, in fact, the same thing called with a different syntax. A function call is merely a procedure call whose final argument is omitted; the value of that argument after the call is taken to be the value of the function call. A procedure call is merely a function call with one extra argument whose value is the value of the function call. So regardless of whether something is defined as procedure or function, it may be called as either an expression or a statement.

Each procedure argument, may be input, output, or both, according to the `?` and `!` annotations on its declaration. A particular ordering of inputs, outputs, and in/out arguments is called a procedure *mode*. A given procedure name may have any number of different declarations with the same argument types and different modes, but may not have more than one declaration with the same types and modes. There are some restrictions on modes specific to functions:

1. if the last argument is output, at least one other argument must be input; and
2. if the last argument is input, at least one other argument must be output or in/out; and
3. the last argument must not be in/out.

If any of these restrictions are not met, the procedure cannot be used with a functional syntax in that mode. These restrictions are imposed to ensure functional syntax is only used for procedures that behave functionally.

As a syntactic convenience, funcs and procs can use patterns in place of formal parameters. This is equivalent to a variable parameter, with a deconstruction statement inserted at the front of the body. Further, funcs and procs can be defined by pattern matching; this involves writing multiple definitions of the same func/proc name and arity, with mutually exclusive patterns in the header. This is semantically equivalent to a single definition whose body is a disjunction of the bodies of all the definitions.

7 Resources

8 The Expression Problem

Wybe solves the "expression problem" by allowing types, funcs, and procs to be declared `open`. An open type is declared by preceding a type declaration with the modifier `open`. An open type in one module can be extended with extra constructors in another module. This is done with an extend declaration, which has the same syntax as a type declaration, except that it substitutes the keyword `extend` for `type`. The type specified in an extend declaration must have previously be declared an `open` type. In this case, the constructors defined in the extend declaration live in the enclosing module, although the type itself may live in a different module.

Similarly, a func or proc may also be declared open by preceding the (first) declaration of the func/proc by the keyword `open`. An open func or proc may be extended by a definition in another module by preceding the definition by the keyword `extend`. Only the first definition of a func/proc need use the keyword. The semantics of the func/proc is then the disjunction of the original and extension definitions. The order of definitions within a module is the order they are written in that module; the modules themselves are taken in an undefined order that is consistent with a module dependencies. That is, if module A depends on B, but not vice versa, then A's definitions must come before B's.

A func or proc definition, whether open or not, and whether the original or extension definition, may also be declared as a default definition by preceding it with the keyword `default` (in place of the `extend` or `open` keyword). This means that that definition is disjoined after all non-`default` definitions. If multiple default definitions are made for the same func/proc, all the defaults for a given module are taken in the order written. If multiple module specify defaults for the same func/proc, then they are taken in the reverse order of the non-default definitions. That is, if module A depends on module B, then B's default definitions come before A's.

9 Foreign interface

10 Code Transformation

Wybe code is transformed into deterministic clausal form as the compiler's intermediate representation. In this form, each procedure body is represented as a (possibly empty) list of primitive operations (comprising the common initial operations for all clauses), and optionally a variable on which to switch and a list of procedure bodies. The value of the variable then determines which (exactly one) of the bodies is executed.

Each operation in a body may have any number of inputs and any number of outputs, but not in/out variables. Inputs may be either variables or manifest constants; outputs must always be distinct fresh variables. For convenience, variable names have an explicit "version" number suffix, similar to the common variable naming scheme used in SSA languages.