n3zqx2l

{A Programming Language Designed By Daniel Rehman}
(03.22.19 – CS 355 – Sp2019)
By: Daniel Rehman

Programming is one of the few occupations which is just as much an art form, as it is an engineering discipline. This is evidenced, in part, by the fact code is not only written, not only read, but run. This property endows things whose goal is to express computer programming, commonly called “programming languages”, a certain set of constraints, which when followed, allow programmers to program a computer lucidly, efficiently, and easily.

Traditionally, programming languages have multiple parts, not unlike spoken natural languages:

- **Syntax**, the *grammar* of sentences in the language
- **Semantics**, the *meaning* of sentences in the language
- **Vocabulary**, the *words* readily accessible in the language

However, any seasoned computer programmer knows that there is one obvious difference between programming languages and spoken languages, and that is in the syntax. The syntax of traditional programming languages is vastly different from spoken English, and as such readability necessarily decreases, if you do not have years of experience using that particular programming language, and its syntax.

```php
if(($fQ=strpos($Fid,'"'))!==false){
    $Fname = sanitize(substr($Fid, $fQ+1, strpos($Fid,'"',$fQ+1)-$fQ-1));
    $p_offset = ($p_close < $p_open) ? $a-(strlen($part_content)-$p_open) : $a;
}
```

Ex. 0: Example syntax in PHP

Needless to say, there are very few English-speaking people who aren’t programmers, who would consider the above (PHP) programming language syntax perfectly readable.

Recently (in the last decade or so), however, there has been a trend for programming languages to become readable in the manner of spoken English, by changing the syntax of the language to match that of the syntax of English. The main examples of these types of languages currently include Python and Ruby, whose code examples are widely regarded as quite readable.

```python
done = False
with open('file.txt') as file:
    for line in file:
        if line and not done:
            print line
```

Ex. 1: Example syntax in Python

Note: To be perfectly fair, the PHP code is for doing a task possibly more involved than the task the Python code is doing, but hopefully the differences between the language's core syntax is apparent and decides its readability, not the task.

But there is one huge problem with programming languages such as Python or Ruby, and that is *efficiency.* Neither Python nor Ruby are notorious for being efficient languages, meaning code written in these languages tends to be quite slow compared to other other languages such as the C programming language, which tends to be almost as fast as hand-written assembly. This is because Python and Ruby are Interpreted languages, as opposed to C which is a compiled language.

To give a feel for the difference this makes, a typical Python program might run as much as *10x slower* than the equivalent C program- and this only gets more detrimental as code size increases. *Thus, because software lives in the real world,* ***Efficiency matters.***

Furthermore, another problem with these two programming languages, Python and Ruby, is their strong "dynamic" tendencies. For example, in Python, some programming errors that would regularly be caught at compile-time, (such as a data type mismatch), are delayed until run-time, causing programs to crash more often than languages which caught that error at compile-time.

In fact, a language where a data type mismatch error is delayed until runtime is regarded as being "dynamically typed", and although dynamic languages are posited as being easier to understand at first, this language property inevitably leads to code which is increasingly unstable and harder to reason about as it becomes more complex.

The problem with having dynamic tendencies doesn't stop at being dynamic typed, but extends to almost any possible error in the code. *If possible, it is* ***better to catch an error at compile-time****, than run-time, always.*

So if you are new to programming, and want a language where you get great performance, and errors caught at compile-time, you might be wondering- why doesn't everybody just code in C? well, the short and simple answer is: *abstractions.* C is notoriously bad at allowing the user to create higher level abstractions which can allow the programmer to think about the problem easier, and thus write better code.

However simply haphazardly allowing for any abstraction in the language won't do, especially if those abstractions result in a loss in performance or efficiency in the language, when used. Ideally, a programming language which could allow user to specify so-called "zero-overhead" abstractions would be the best combination, allowing the programmer to think about the problem better, while still maintaining great performance and efficiency. *This is because* ***abstractions are key to writing readable, reliable software****.*

Luckily, a programming language which employs these zero-overhead abstractions for the most part already exists, and it's called C++. C++ is well known to allow for the usage of very efficient, lightweight abstractions, which collapse at compile-time down to efficient assembly code. However, C++ is also well known for being an *arduously* complex language, whose syntax is not always the most intuitive, nor the most readable. For example, this syntax leads to C++ code similar to:

```
1 vector(size_type __n, const_reference __x, const allocator_type& __a);
2     template <class _InputIterator> vector(_InputIterator __first,
3             typename enable_if<__is_input_iterator  <_InputIterator>::value &&
4                                !__is_forward_iterator<_InputIterator>::value &&
5                                is_constructible<
6                                   value_type,
7                     typename iterator_traits<_InputIterator>::reference>::value,
8                                _InputIterator>::type __last);
```

Ex. 2: C++ excerpt taken from std::vector in libc++

It is not an audacious claim to say that for most programmers who have not worked with C++ closely for years, this is not completely natural to them, to say the least.

When it comes to syntax, there are specific lessons on what *to do*, from languages like Python or Ruby, as well as specific lessons on what *not to do*, from languages like C, C++, PHP, Perl, and a host of other languages with non-intuitive syntax. From the point of view of reading and maintaining code, (which is done far more often than writing code, on average), the readability and intuitiveness of the syntax has a large effect on how maintainable code is in general. *This points to the final key principle, which is among the most important:* ***readability counts.***

In summary, the following principles are deemed necessary and thus important to possess in a programming language:

0. Efficiency matters.

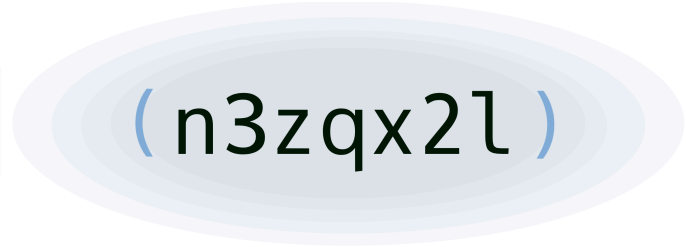
1. Compile-time Errors are better than run-time errors.

2. Abstractions are key to writing readable, reliable software.

3. Readability counts.

So, you might be wondering: ***is there a language which brings together all of these design principles into one unified, amazing programming language?***

Yes there is. and It's name is `n3zqx2l.`



`n3zqx2l` (all lowercase) is a programming language created by the author, which is a **minimalist**, **multi-paradigm, efficient, strongly, statically and nominally typed**, **statically**/**lexically scoped**, **aggressively type-inferred, expression-based**, **compiled, JIT-compiled/interpreted** language designed with ***code readability*** and ***runtime performance*** in mind, and utilizes a *novel signature system*, *compile-time abstraction evaluation with dependent types,* ***and is built on top of the LLVM compiler infrastructure.***

Before diving into any further detail about the language and its syntax, heres the canonical “Hello World” program:

```
1  ()
2      use .io
3      print "hello, world!"
```

Ex. 3: The Hello World program in `n3zqx2l`.

Believe it or not, theres actually a lot going on here. But first, something very important to note, is that “`use`” and “`print`” are not keywords (despite the syntax highlighting), or even builtin the language in any way. They are both actually *abstraction calls*, to abstractions defined in the core standard library (also known as “*`_core`*”), which is being implicitly imported at the top of the file, before line 1.

In fact, not only are there are **no keywords** found in the language, but there are also **no builtin mathematical operators**, either. There are no “`for`” loops, or “`while`” loops, “`if/else`” statements, or really, anything. There are only *abstractions,* and *variables* (and some lower level stuff, which will be covered later).

Note: These things called “*abstractions”* are the most important thing in the language, so you will probably hear a lot more about them.

The two parenthesis on line 1 are declaring an abstraction, which takes no arguments and has no signature (meaning its anonymous), which by convention of the language, and the fact its at the top level, means that this is the entry-point for the executable: in other words the “`int main(void)`” function in C-like languages. Obviously just like “`int main(void)`”, you can only have one top-level anonymous abstraction.

An important note about the call to “`use`”: it not only serves to include the “module interface” file “`_core.io.ni`”, but also to allow its direct children to be used in the current namespace. This is the reason why we weren’t forced to say “`io print`” on line 3.

Also, notice the indentation! Indentation has semantic meaning in this language, *only* when curly braces aren’t used. When creating any block of code, you can always choose between **using indents** (by default, of 4 spaces, however tabs are also accepted) and no curly braces, or **using curly braces**, and having indentation simply be ignored by the compiler, for that block/level of code.

Finally, there *are* no semicolons at the end of the lines, because first, newlines are sufficient, as its impossible to put two statements on the same line in this language anyways, and second, semicolons actually denote comments (both single-line, and multi-line) in the language, as shown in the below code examples.

More code examples!

```
;. (obviously you would never write something like this.) ;

(add first number and second number together) {
    first number + second number    ; simply return the sum!
}
```

Ex. 4: An example abstraction definition, with comments.

The above (silly) code example serves to demonstrate `n3zqx2l`'s almost exclusive use of "***space case***", as opposed to the traditionally used "*snake_case*", "*camelCase*", or "*kebab-case*". Spaces can be found in variable signatures or abstractions signatures, and in general, the idea of "juxtaposition" meaning anything specific (eg, function application, etc.) all the time, does not exist in this language- spaces are just part of the signature, and they work just like you'd expect.

Just to get this out of the way, the syntax for comments is simple: *single-line comments* are denoted by a **semicolon followed by a space**, and go until the end of the line, while *multi-line comments* are denoted by **a semicolon followed by anything *but* a space**, and is terminated by the nearest semicolon. Although comments really shouldn't be necessary in this language, as you will see later on.

In the signature "`add first number and second number together`", notice that parts of the signature (namely "`add`", "`and`", and "`together`") are not only allowed to appear before parameters, (ie, "prefix" notation, as in other programming languages), not only allowed to appear after parameters, (ie "postfix" notation), but also *between* the parameters *(infix)*, or indeed any possible combination of the above, for any number of parameters from 0 to however many can fit onto the stack.

> *Note, that despite there already being an abstraction named "`… and …`" in the standard library core, the compiler won't get confused or parse in correctly when we try to call "`add … and … together`", because* ***the compiler will always call the signature with the longest name.***

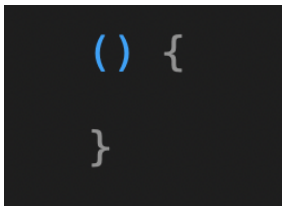
As if this wasn't enough, you can put just about any other printable ASCII or Unicode symbol inside a signature, as long as it isn't one of "`:`", "`(`", "`)`", "`{`", or "`}`", which are the only reserved operators in the syntax of the language. Every other symbol is fair game to be found in signatures. Only alphanumeric (or "_") characters require spaces between them, to separate distinct identifiers- in other words, signatures work just like you'd expect.

Note: there is one slight constraint to signatures, however: any particular alphanumeric sequence must not start with an underscore, as those identifiers are reserved by the compiler. This includes reserved identifiers like "`_`", "`_identifier_Here`", "`_1234_1234`", and "`__`" but not unreserved identifiers like "`a_`", "`123_identifier_Here`", "`1234_1234`", "`2a`", or "`5`". (yes, "5" is a valid identifier).

> *So essentially, (except for underscore rules) the signature can take on **any** form, and thus its best to simply choose the signature that **fits best for your problem** or abstraction, and thus is most readable. **Readability is the goal here.***

A note about these so called "abstractions": Abstractions are called so for a reason. This is because, technically, they are a **function**, a **procedure**, a **namespace**, a **record**, a **struct**, a **class**, a **union**, an **enum**, and just about any other typical builtin programming language abstraction, all in one- built into a single syntactic construct: ***the abstraction.***

This, besides decreasing the complexities of the language and thus the amount the programmer needs to learn, also frees the programmer from unconsciously restricting their solutions to only the linguistic features commonly found in other programming languages, and thus possibly obtaining an ineffective, or less readable solution.



Ex. 5: An anonymous empty abstraction, which takes nothing, and returns nothing.

(aka, the simplest program in `n3zqx2l.`)

Note: This references the "Sapir-Worf Hypothesis", which states that the language that you speak inevitably dictates how you think about things. This language aims to apply that principle to programming languages, and free the programmers thinking by reducing the number of linguistic/syntactic/semantic constraints.

Finally, here are two code examples, the first of which is factorial program, and the second of which is a simple prime generator. Details of these examples (such as the back tic string, the core standard library usage, and other details will be explained in the next section.

```
;. yes, unicode characters are allowed
   in signatures (the "≤" abstraction
    is an alias for "<="). because why not? ;

(n: num !)
    1 if n ≤ 1, else (n - 1)! * n

()
    use .io
    for n in 5, print n!

;. output:

   bash$ nostril factorial.n -named factorial
   bash$ ./factorial
   1
   1
   2
   6
   24
   120
   bash$
;
```

Ex. 6: A simple factorial program, with its output shown in the comment below it.

```
; a prime number generator in n3zqx2l.


` this is a function which returns
 whether a given number is a prime number.`

(n is prime)
    if n <= 1, return false

    use .math
    for i in 2 to sqrt n,
        if n is a multiple of i, return false

    true


(args: [string])
    N = args[1] as num

    use .io
    if n is prime, print n, for n in N
```

Ex. 7: A prime number generator, which gets a number from the command line arguments.

More Language Details!

However, if you have programming background, you might be scratching your head thinking *"how on earth is the compiler supposed to parse that definition?*" The short answer to that is ***time.***

> *This programming language makes an **explicit**, **intentional** tradeoff between **fast compilation time** and **readability of code.** Specifically, this means the language has generally slower compilation times than other languages, while being magnitudes more readable than other programming languages.*

Behind the scenes, however, the method the compiler is employing to allow for such permissible syntax and signatures, is an algorithm called "call signature resolution" (CSR), which is a simple algorithm devised by the author.

Before *this* step, however, there are many other stages that must be completed which build up to CSR, including but not limited to, ILR, TIB, ETA, ETV, SVA, NSS, TIC, NVS, etc. (these are all stages in the compiler, but which are outside the scope of this document.) Having all of these stages makes the front end of the compiler, at first glance, seem somewhat similar to in style to a *nano-pass compiler,* which isn't necessarily great for speed, but aids in reliably, **correctly parsing complex input.**

One Interesting detail about the front-end of the compiler is that **its actually an API.** `n3zqx2l` code can make calls to intrinsic abstractions (also called *"builtins"*), defined by the compiler, to for instance, modify the symbol table, recognize a particular Abstract Syntax Tree (AST) node, or even perform compile-time evaluation of arbitrary expressions.

Builtin signature elements always start with an underscore character, and don't use any non-alphanumeric/underscore characters. Examples of existing builtins include:

```
(_define ;signature; _as ;type; _in ;scope;)

(_undefine ;signature; _in ;scope;)

(_parse _expression)

(_do ;expression; _at _compiletime)

(_do ;expression; _at _runtime)

(_precedence ;integer;)
```

Ex. 8: example of some builtin signatures in `n3zqx2l`

And so on. These builtins serve to allow user code to *inform the compiler at a very fine-grained level*, what to do when compiling their code, whether that be during parsing, analysis, compile time evaluation, or any other phase with builtins. Because of this freedom in how code is compiled, the language can be extremely minimalist, and only include what is absolutely necessary. Interestingly, one such thing which turns out to *not* be necessary in this language, is builtin math operators.

You may be asking, however, "Then how can the language be efficient at all, if there aren't even builtin math operators?!" Well the answer to that is quite simple: **LLVM.**

`n3zqx2l` and LLVM:

LLVM strings are one of the 3 builtin string types in the language, (the others being **text strings** and **documentation strings**, more on those later.) The LLVM Intermediate Representation (IR), which is the language used by the LLVM compiler infrastructure, (and thus used by `n3zqx2l`), essentially forms the low-level efficient computational base of the language, and allow `n3zqx2l` to be very **high level and minimalist**, while **still being able to manipulate the hardware as fast as C code**.

The syntax of LLVM IR is outside the scope of this document, however, if you are unfamiliar with the IR, and wish to learn more, there are a plentiful amount of resources online
Here's an example usage of LLVM strings in `n3zqx2l`:

```
1   (a: 'i32' + b: 'i32') 'i32': 'alwaysinline' {
2       '%"my result" = add i32 %a, %b'
3       return my result
4   }
```

Ex. 9: An efficient implementation of "+" for 32-bit integers, which uses LLVM strings.

In fact, this definition of "+" is quite similar to the one found in the *core standard library* which defines a lot of these base operations for you, (so you don't have to define them yourself.) Note that LLVM strings can appear just about wherever an expression can appear, however when found in a place expecting a type, they must evaluate to some sort of LLVM type, (e.g., '`i32`') and when found in a place expecting an abstraction type, they must evaluate to a valid LLVM function attribute(s), (e.g., '`alwaysinline`').

Another important detail about LLVM IR in this language, is that variable and abstraction signatures can pass freely between the different languages, as line 2 and 3 demonstrate. Variable names simply transfer by putting the (space-containing) signature in double quotes. However, the conversion into LLVM IR is even easier if there are no spaces, *simply use the variable name directly*, as shown in lines 1 and 2.

> *As you can hopefully see, though, LLVM IR can be intermixed with `n3zqx2l` code quite easily, and thus can be composed together to form **extremely fast and efficient, lightweight, readable abstractions.***

As it turns out, `n3zqx2l` and LLVM IR are actually not too different as programming languages, (ignoring the obvious difference in abstraction level, that is.) For example, here's a small comparison between the two:

	LLVM IR	`n3zqx2l`
Mutable?	SSA form	All values are `const` by default
Typed?	Strongly typed	Strongly typed
Scoped?	Statically scoped	Statically scoped
Garbage collected?	No (unless its configured)	No
Nominal?	Yes (for identified types)	Yes

Ex. 9: A brief comparison between LLVM IR and `n3zqx2l`.

So essentially, the take-away is that despite them seeming quite different, LLVM IR and `n3zqx2l` are actually quite similar, and thus *are great companion languages.*

Abstraction Call and Type Signatures:

Interestingly, there is an aspect of the language which is important to get to know when transferring abstraction signatures into LLVM IR, or doing other tasks: the idea of an abstraction's **call signature** and **type signature.**

```
; definition                ; call signature        ; type signature

(f x: a y: b) c {               (f a b)                   (a b c)
    ;...;
}
```

Ex. 10: Example call and type signatures for a given abstraction definition.

As you can see, Its fairly intuitive: assuming you have types "a", "b", and "c", you simply take the abstraction's definition signature, (e.g., "`(f x: a y: b)`") and for each parameter (e.g., "`x: a`", you put the parameter's type signature ("a") in replace of the parameter. That gets you get the *call signature: "*`(f a b)`*".* To get the type signature, you simply make a Lisp-like list of the parameter types, and stick put the return type on the end. For our example, we get: "`(a b c)`". Note that the parenthesis are actually required, when specifying a call signature or type signature. Without them, they would simply interpreted as a abstraction call.

One caveat: if you are trying to specify the call signature "`(f a b)`", but there's already an abstraction called "`(f a b)`", then you can disambiguate between a parenthesized call to that abstraction, and the call signature you want, by writing "`(f (a) (b))`". Same goes for type signatures. However, this isn't usually necessary, however, because usually you don't need to put calls in parenthesis very often, as you will see later.

Finally, calling an abstraction defined in `n3zqx2l` code, from LLVM IR code is fairly simple now:

```
; some random function:
(f x: int y: int) int {
    return x + y * 2
}

()
    g = 'call %int %"f (int) (int)"(%int {1}, %int {1})'
    ; g is an "int", with the value 3.

    ; (obviously "g = f 1 1" is more succinct)

```

Ex. 11: Example call to a `n3zqx2l`-defined abstraction, from LLVM IR.

In other words, simply put parenthesis around the types present in the conventional call signature for the abstraction, and put the whole thing in quotes.

Compile-time Abstraction Evaluation:

One of the unique features of n3zqx2l, which few other languages have really accomplished, is the seamless transition between writing code that will run at compile-time, versus runtime. Compile-time abstraction evaluation (CAE) is fundamental aspect of n3zqx2l, and allows code to post-pone as little code until runtime as possible, aiding runtime performance.

As shown in an earlier section, there are two builtins which are rather useful for specifying when a computation should be done, namely:

```
(_do ;expression; _at _compiletime)

(_do ;expression; _at _runtime)
```

However, there are actually two more, which are used in signature types, (the type of an abstraction), in order to explicitly mark the abstraction as either evaluation type. These are:

```
(_compiletime)

(_runtime)
```

These two are useful when you have a function which is normally evaluated at runtime, and you want it to be done at runtime, or vice versa. However, these do not need to be used in order to achieve CAE. In fact, **all abstractions are actually by default `_compiletime`,** and only when the compiler can't prove that the body of the abstraction is able to be run act compile-time, does it revert the abstraction to being marked runtime. This is useful because it means that the compiler is essentially actively checking which parts of the code can be done at compile-time, without the user having to lift a finger.

However, in cases where this would not be desirable to do this, (for instance, if the computation is very intensive and is better left to runtime, even though it *can* be done at compile-time) the user can use the above "`_runtime`" and "`_compiletime`" builtins, to specify the evaluation time explicitly, like so:

```
(do something crazy) arbitrary int: _runtime {
    return 42 + 1234567 * 98989 ^ 2
}
```

Ex. 12: Example usage of builtin "_runtime" to change abstraction evaluation time

Nested Abstractions:

This is a particularly important part of the language: the ability to not only compose abstractions calls, but nest their definitions entirely. This is what gives abstractions their "namespace/struct/class" quality, as opposed to just being a fancy name for a function or procedure.

How nested abstractions work is best explained via example, rather than words, so here's a short example of it in action:

```
(thing) {
    inside _: writeable int = 0
    _ data: readable int = 0

    (default _) _ {}

    (make _ better) {
        _ data = 5
    }
}

() {
    my thing: default thing
    make my thing better

    use .io
    print my thing data
    inside my thing = 1
}
```

Ex. 13: Example of nested abstractions, and their usage.

So what is actually happening here?

In short, its all based on the single "_", which is semantically similar to the "this"/"self" construct in other object-oriented (OO) languages. One important difference, however, is that "_" isn't just used for OO programming. "_" is also used for simply making namespaces, or organizing abstractions in general.

> *Behind the scenes, the compiler is actually doing an algorithm called "Namespace Signature Substitution" (NSS). All it does is simply* ***substitute the parent abstraction's signatures, in replace of "_"'s, found in sub signatures.***

One exception to this, however, is when code instantiates a particular abstraction, in which NSS will substitute the signature of the variable instance in replace of the "_", as opposed to the abstraction's signature itself. This is evident in the first line of code in the entry point of the above example.

Notice that in the above example, we create an instance of the "thing" abstraction, called "my thing", but we do so by calling the function "(default _)" which seems to return "_". Returning "_" semantically means that this abstraction is a "constructor", in other words it returns a new instance of the abstraction.

In a sense, this is quite similar to an object in an OO language, with one difference: code can appear inside the definition of the abstraction, and thus is run when the object is instantiated. This is similar to how functions work in Javascript, to some degree.

Finally, one of the great things about variable and abstraction signatures (and how "_" can appear in them) is that signatures end up composing in quite readable ways. For instance, because multiple "_"'s can appear within a signature, we get the following syntax for a very useful operation:

```
; t is not defined, thus is a generic type.
(vector t: _type) {
    _ data: *t

    (_ [ _ > threshold: t ])
        result: vector t
        for i in _ data, if i > threshold, result += i
        return result

    ; *constructor and destructor ommited*
}

() {
    ages: vector = [12, 34, 6, 46, 67]

    old ages = ages[ages > 30]
    use .io
    print old ages                ; prints "[34, 46, 67]"
}
```

Ex. 14: A testament to how powerful well-made signatures can be.

Miscellaneous Stuff:

There are a couple other interesting features of this language, which don't really fit under any particular category, but are important for the language's usability.

The first of these is known as **"user-defined precedence",** and **"user-defined associativity"**, which is basically exactly what it sounds like- it is a way for the programmer to actually specify the precedence and associativity for a particular abstraction they are making. This is simply done by using "`(_precedence ;integer;)`" builtin, where a negative integer means that it has a lower precedence than the default for an abstraction, (which is "`0`"), and a positive integer means it has a higher precedence than the default for an abstraction. Likewise, the builtin "`(_associativity ;_handedness;)`" accepts one of two also builtin abstractions, which are "`_right`" or "`_left`", which mean exactly what you think they would. These two builtins help reduce the amount of parenthesis needed in expressions.

Another thing that is quite important about the language, is its ability to interface with the C standard library, via the language's Foreign Function Interface (FFI). This is essentially achieved with very little effort, due to the fact this language is built using the LLVM compiler infrastructure, which can allow IR generated by `n3zqx2l` code to call outside precompiled C functions, as long as they are linked properly. This will most likely form the base of the standard library, when it is written

A very important note about this language, is that although the `n3zqx2l` compiler/interpreter, known as "`nostril`" is almost operational, "`_core`", the core standard library of `n3zqx2l`, (which defines mathematical operations, for loops, if statements, typeless variable declarations, and a host of other fundamental/useful programming constructs) is not yet written, and thus the language is not yet ready for production/serious use.

However `_core` will be written soon!

Closing thoughts:

After getting used to the permissibility in creating signatures in this language, I think you will find that you will have a hard time going back languages which force a single prefixed identifier for a function or variable name, which cannot include any operators at that. It is honestly a shame that programmers are forced so often to write:

```
bool is_divisible_by(int x, int y) {…}
```

When it is so incredibly obvious that what we wanted to say is:

```
(x is divisible by y) {…}
```

And considering the parsing isn't even that complicated if you get all your ducks in a row, (it just takes longer) there is no excuse for a supposedly "high-level" programming language making you write anything like the first version.

My hope as the author of this language is that you find using `n3zqx2l` to be a liberating experience as a programmer, freeing not only your *syntactic expression*, but also *how you think about programming* on a deeper level, and eventually free and empower you to write ***expressive, efficient software.***

If you are curious about why the language is called "`n3zqx2l`", its because it had 0 search results on Google, and apparently the designer of the language can choose whatever name they want. So I picked something nice and memorable.