The Nex Programming Language

The Nex Programming Language is a passion project to learn more about compilers and assembly language. Nex Lang is a compiled programming language, which means that Nex source code is translated directly into assembly language.

As this is my first time writing a compiler from scratch, keep in mind that some features might not be implemented in the most effective or efficient way possible. I welcome feedback, contributions, and suggestions from anyone interested in language design.

Nex Lang is entirely open source. You can access the GitHub repository through my personal website, which also provides additional information.

For those interested in building the compiler, assembling the output, or experimenting with the language, more information is provided in the GitHub repository.

I also wanted to mention that this guide is designed to be read in a chronological order.

Variables

The only type of data that can be stored in Nex Lang is an 8-bit unsigned integer. These are statically allocated, and all variables are global.

Variables can be declared with the ‘var’ keyword, followed with a variable name. From this point, you would generally follow with an equals sign and an expression. Additionally, you should probably know that all statements in Nex Lang end with a semicolon.

There is no support for type annotations or type inference, as the type system is intentionally minimal.

```
var foo = 20;
var bar = foo;
```

We declare a variable called ‘foo’ and we set it to a value of 20. Afterwards, we create another variable called ‘bar’ and we set its value to the value of foo (20).

Variables can also be declared without the equals sign and without a value. This automatically sets the variable to a value of 0.

```
var foo;
var bar = foo;
```

We declare a variable called ‘foo’ and set its value to 0 by lacking an equals sign and value. Afterwards, we declare ‘bar’ and set it to the value of foo (0).

Exit

Exit statements can be specified with the ‘ex’ keyword.

Exit statements contain an expression. This expression is evaluated (using assembly language), and its result is used as the program's exit code.

Like all statements in Nex Lang, exit statements must end with a semicolon.

The expression inside an exit statement must evaluate to an 8-bit unsigned integer for intended functionality.

```
var foo;
ex foo;
```

This creates an exit call with the value of foo (0).

The above can be easily represented in assembly language.

```
mov rax, 60
mov rdi, 0
syscall
```

The ’rax’ register is used to store 60, which is the syscall number for exit. The value that we want to exit with (0) is stored by the ’rdi’ register. After that, we use 'syscall' to system call and exit with a value of 0.

Math

Nex Lang has four mathematical operations: addition, subtraction, multiplication, and division. These are performed using standard arithmetic symbols.

Additionally, Nex Lang supports operator precedence, meaning multiplication and division are evaluated before addition and subtraction.

Operator	Precedence level
+ (Addition)	1
- (Subtraction)	1
* (Multiplication)	2
/ (Division)	2

The language does not allow the user to control precedence using brackets or parentheses.

All results are constrained to 8-bit unsigned integers, therefore results that go above 255 will wrap around.

```
var foo = 50 + 2 * 5 - 6;
var bar = foo - 9;
ex foo - bar + 1;
```

The value of ‘foo’ will be equivalent to 50 + 10 – 6 = 54. This is because the 2 * 5 will be evaluated first. The value of ‘bar’ will be 54 – 9 = 45. The program exits with value 54 – 45 + 1 = 10.

Comments

A comment in Nex Lang is simply a part of the source code that the Lexer skips, meaning that it is ignored.

There are two types of comments – single-line comments and multi-line comments.

Single-line comments begin with a hashtag symbol and end at the newline. Multi-line comments begin with a hashtag symbol followed by a dollar symbol and end with a dollar symbol followed by a hashtag symbol.

```
# this is a single line comment

#$ this
is
a
multiline
comment $#
```

Obviously, if you place comments in the middle of certain lines, then the code will not compile, and you will get errors.

```
var x = # this is wrong 10;
```

The exception is multi-line comments. If you correctly place a multi-line comment in the middle of a line, the code will work perfectly fine.

```
var x #$ this is right $# = 10;
```

Arrays

Arrays in Nex Lang are basically a collection of variables, therefore have the same restrictions as them. For example, all arrays are global, just like variables.

All arrays in Nex Lang always start from zero.

Arrays can be declared with the ‘arr’ keyword. This is followed by a pair of square brackets, with the size of the array within the brackets.

The size cannot be an expression, meaning it must be a constant value. Like variables, arrays are statically allocated and cannot have variable sizes.

As always, a semicolon is required to end the statement.

```
arr array[50]; # size of 50
```

Arrays can also be declared without a size within the brackets. In this case, the size of the array will automatically be set to 30,000.

```
arr array[]; # size of 30,000
```

This will also be the case if you declare an array without any brackets at all.

```
arr array; # size of 30,000
```

An unassigned index of an array might contain a non-zero garbage value. The only way that you can assign values to a specific index of the array is through reassignment, which we will cover in a later section.

Set

Set statements in Nex Lang are very similar to setting labels in assembly language.

A label is set using the ‘set’ keyword. Next, you provide the name of the label that you want to set. The statement ends with a semicolon.

It’s important to note that labels are global in Nex Lang, much like variables, and can be accessed throughout the program.

```
set label; # sets a label called ‘label’
```

This can be directly represented in assembly. Here, one line of Nex Lang code correlates to one line of assembly, making the conversion 1:1.

```
; sets a label called ‘label’
label:
```

Set statements are incredibly useful as they help us jump to specific parts of the program.

These labels allow for better control over the flow of execution, as you can jump between different parts of the program as needed.

This becomes important when implementing more complex logic or handling repetitive tasks.

In the next section and later in the guide, we will explore how to use set statements, along with go statements to create loops.

Go

Go statements in terms of syntax are very similar to set statements.

A go statement is defined by using the ‘go’ keyword. It is followed by the name of the label that you want to go to. Next, a semicolon is needed to end the statement.

Go statements allow the program to jump to a specific point defined by a previously set label.

Please note that to go to a label, it needs to be set at some point in the program. The compiler does not check if the label exists before trying to go to it.

Therefore, it is good practice to make sure that the desired label is set before using a go statement.

```
go label; # goes to ‘label’
```

A go statement is very similar to a set statement and therefore can also be represented 1:1 in assembly language.

```
; goes (or jumps) to ‘label’
jmp label
```

As explained previously, both set and go statements are very important for implementing control flow in Nex Lang.

They are useful for creating loops, and other control structures which are covered in later sections of the guide.

Reassignment

For reassigning variables, you specify the identifier of the variable you want to reassign. Follow that up with an equals sign, and then the new value (or expression), followed by a semicolon.

The new value must be an 8-bit unsigned integer. You cannot reassign variables to types other than the supported 8-bit integer. The compiler does not check this.

```
var x = 10; # originally sets x to 10
x = x + 5; # reassign x to 10 + 5 which is 15
ex x; # exits with 15
```

To reassign a specific index of an array, you first specify the name of the array, followed by the index (which can be an expression) that you want to reassign within square brackets.

Next, an equals sign is needed, followed by the expression that will be evaluated and stored at the index specified. Finally, you need a semicolon to end the statement.

The new value must be an 8-bit unsigned integer. The compiler does not check this. The compiler also does not check if the index is within bounds, so going out of range means that you could modify the stack.

```
arr x; # declares array x with size of 30,000
x[5 + 2] = 10; # reassigns index 5 + 2 of the array to 10
ex x[5 + 2]; # exits with 10
```

You can also do math with the values stored at specific array indexes. They do not have to be in the same array, but in this example, they are.

```
arr x; # declares array x with size of 30,000
x[8] = 20; # reassigns index 8 of the array to 20
x[5 + 2] = 10; # reassigns index 5 + 2 of the array to 10
ex x[5 + 2] + x[8]; # exits with 30
```

Ifz

An Ifz statement is specified with the ‘ifz’ keyword. After that, you specify the expression you want to test.

A statement body is constructed using curly braces. An open curly brace marks the start of the statement body, and a closed curly brace marks the end of the statement body.

If the expression evaluated is equal to zero, the code inside this block will execute.

The body of the Ifz statement can contain multiple statements, and these will only run when the condition is met. In this example, we only have one statement inside the body.

```
arr x[10]; # declares array with size of 10
x[1] = 5; # sets index 1 of array to 5
var c = x[1] + 5; # declares var c and sets to 10

ifz x[1] {
    ex 15;  # doesn’t execute as x[1] is not equal to 0
}

ex c; # exits with 10
```

We can try the same code but set x[1] to 0 this time.

```
arr x[10]; # declares array with size of 10
x[1] = 0; # sets index 1 of array to 0
var c = x[1] + 5; # declares var c and sets to 5

ifz x[1] {
    ex 15; # exits with 15 as x[1] is 0
}

ex c; # does not exit with 5 as we already exited above
```

El

An el (else) statement needs to have a corresponding Ifz statement above it.

An el statement can be created using the keyword ‘el’. An el body is created in the same way that an Ifz statement body is created, using curly braces to define the start and end of the block.

The code inside the el block will execute only if the expression is not zero.

Just like the Ifz body, the el body can contain multiple statements, and it is only executed when the expression is not zero.

We can utilise Ifz, el, set and go statements to create cool loops.

```
arr x[10]; # declares array with size of 10
x[5] = 10; # sets index 5 of array to 10
set before_loop; # sets a ‘before_loop’ label
ifz x[5] {
    ex x[5]; # if x[5] is 0, exit with value of x[5] (0)
} el {
    x[5] = x[5] - 1; # if x[5] is not 0, sub 1 from x[5]
    go before_loop; # go back, before the ifz and check again
}
```

As you could see above, loops with a limit are completely possible, as well as loops without a limit due to functionality that allows an infinite set and go cycle.

You can control the number of iterations by incorporating control flow techniques using Nex Lang features.

In

The in (input) is treated as an expression. This means that you can assign variables and indexes of arrays to the value of in, and it will store the input from the user.

It takes in a single character from the user and converts it to an integer according to an ASCII table.

When the user provides input, in reads that input, converts it to an 8-bit unsigned integer, and assigns it to the variable or array index you specify.

```
ex in; # get value from the user and exit with it
```

You can also store the value of the input in variables.

```
var x; # declare variable, set to 0
x = in; # reassign x to user input
var z = in; # declares variable, set to (new) user input
ex z + x; # adds both the inputs together, might be > 255
```

You can also store the value of the input in specific indexes of arrays.

```
arr array[50]; # declare an array with size of 50
array[5] = in; # store user input at index 5 of array
ex array[5]; # exit with the user input
```

Out

An out statement takes in an identifier name and outputs the ASCII equivalent of its value.

An out statement is specified by the ‘out’ keyword. Next, you need to specify an identifier name of a variable (not an array) that will be outputted to the user.

The variable should resolve to an 8-bit unsigned integer, which will be interpreted as an ASCII character.

Finally, (I think you know this by now) we need to end the statement using a semicolon.

Please note that an out statement only takes an identifier, not an expression, so you cannot output 50 + 50. You would have to store 50 + 50 in a variable and output the variable.

```
out 104; # outputs ‘h’
```

You can also output more complex expressions.

```
var x = 20; # declares variable, set to 20
var k = 100; # declares variable, set to 100
arr i[10]; # declares an array of size 10
i[10] = 5; # sets index 10 of the array to 5
var b = x + k + i[5 + 5]; # declares variable, set to 125
out b; # outputs ascii equivalent of 125 ‘}’
```

This is useful when handling data, especially when you need to process or display information that is represented by numeric values.

It also provides a straightforward way to output characters.