Basics

Types

`lush` is a typed shell. The following types exists: * Any - The type can be of any type * Nil - The empty void type * bool - Boolean, either `true` or `false` * num - A number, e.G. 1, 0.5, -5 * str - A string, e.G. “Hello World” * [] - An array of , e.G. [1 2 3] * Structs - See below * Functions - See below

`lush` supports type inference. Types do not have to be spelled out each and every time - they are mostly inferred due to the usage of variables, constants and commands.

Comments

```
# Everything behind the # until the end of a line, is considered a comment
```

Variables

Variables can be declared via the `let` statement.

```
let unassigned  # Variable without initial value
let x = 1       # Variable with intial value
```

Variables are typed in lush. If the type is not declared, it will be inferred based on the usage.

```
let var1: num = 1 # var1 with an explicit type (number)
let var2
var2 = 2 # var2 inferred to be a number here
```

Type coherence is statically verified. Meaning: there won’t be type errors during runtime.

if - elif - else

```
let lush = 1
if $lush < 3    # The following equality operators are supported: < <= == != => >
    echo Hello
elif $lush == 42
    echo world
else
    echo "!"
end
```

for

Iteration over strings and arrays is possible.

```
for character in "abcde"
    # character is of type string
    echo $character
end

for elem in [1 2 3]
    # elem is of the arrays inner type (number here)
    echo $elem
end
```

Command calls

A command (or function) can be called by typing its name and the arguments.

```
command_or_func_name $arg
```

If the commands name is not found, lush will try to start a process by that name

```
echo $arg # starting the 'echo' process with $arg as its first argument
```

For convenience: when passing simple-words to arguments of type `str`, they do not have to be quoted.

```
echo Hello World "!" # Better quote operators. They are not promoted to strings automaticall
```

Pipes

Commands do not only receive arguments via arguments and flags, but also by what is "piped" into them.

```
echo "This value gets passed to stdin of cat" | cat TODO buggy
```

Structs

Lush has c-style structs. Please note, that struct-names have to (!) start with an upper case letter.

```
struct Ip{ # Declaration
    a: num
    b: num
    c: num
    d: num
}
let x = Ip { a: 192 b: 0 c: 0 d: 1 }
```

Functions

A function can be declared via the `fn` keyword

```
fn my_first_fn
    echo "Hello"
end
```

Return

Functions can return a value via the `ret` keyword.

```
fn func1 (ret: int)
    ret 1
end
fn func2 (arg: int)
    ret $arg # The return type of func2 is inferred to be int
end
```

If the return type of a function is not declared, it will be inferred. However all `ret` statements within a function have to be type consistent.

```
fn fn_with_type_error(arg: int)
    if $arg < 1
        ret "Less than 1" # Type 'str' here
    end
        ret $arg          # Type 'int' here
    end
end
```

Input

Values can be "piped" into a function. Those values can be handled via the special `in` argument.

```
fn take_num_ret_num(in: num ret: num)
    ret $in
end
1 | take_num_ret_num
```

`in` does not have to be declared. It will be automatically received.

```
fn take_num_ret_num
    ret $in
end
1 | take_num_ret_num
```

Arguments

Functions can accept arguments by declaring them within a `signature`

```
fn fn_with_args (arg1: num arg2:str)
    echo $arg1 $arg2
end
```

A variable amount of arguments can be taken by declaring a var_arg argument

```
fn fn_with_args (arg1: num ...rest: num)
    echo $arg1
    for val in $rest
        echo $val
    end
end
```

If no signature is declared a command will have an implicit var_arg argument `args` of type `[any]`

```
fn passthrough
    echo $args
end
passthrough 1 2 3
```

Flags

Flags can be declared by prepeding "–" to their name.

```
fn fn_with_flag( --flag1: num )
    echo $flag1
end
fn_with_flag --flag 1
```

If the type of a flag is not declared, it defaults to bool. Boolean flags are like switches, passing them assigns `true` to them, `false` otherwise.

```
fn fn_with_switch( --switch ) # Type of switch is bool
    echo $switch
end
fn_with_switch --switch # prints true
fn_with_switch          # prints false
```

A flag can also be given a shorter name (one character name), or only a shortname

```
fn fn_with_flag( --flag1 -f: num )
    echo $flag1
end
fn fn_with_short_flag( -f: num )
    echo $f
end
fn_with_flag --flag 1
fn_with_short_flag -f 1
```

Flags are by default optional to pass, but they can be made required by adding the `req` keyword

```
fn fn_with_req_flag( req --flag1: num )
    echo $flag1
```

```
end
```

Function overloading

Functions can be overloaded by their required flags

```
fn file(arg: str
        req --delete )
    rm $arg
end

fn file(arg: str
        req --list )
    ls $arg
end

file --delete file1 # Deletes file1
file --list dir1    # Lists the contents of dir1
```

Function purity

Functions can be marked `impure`. Lets refine the last example:

```
impure fn file(f_to_del: str
        req --delete )
    rm $f_to_del
end
```

Running an `impure` function or command (might) change the state of the machine. When running such a command during a debug session, the debugger will print a warning, asking whether the command shall be executed or skipped. (See the debug chapter reference) External commands are by default considered to be impure, unless their name appears in a list of well known pure external commands (e.G. “cat”, “awk” ... see lush/crates/lu_cmds/src/external_cmds_attr.rs for a complete list). User defined functions are neither considered to be pure nor impure. The debugger will step into them, but will check any command call for its purity before execution.

Generic functions

Functions can have generic arguments. For example the `push` command from the `std:array` module could be visualized in lush code as follows:

```
fn push(array: [T] ...to_push: T)
    # Impl here ...
end
```

Generics provide type safety. The inner type T of “array” does not realy matter. However the values “to_push” needs to be of the arrays inner type T. This can

be statically described and verified by generics.

The name of the generic type cannot be freely choosen. Only T0, T1 ... T9 and U0, U1 ... U9 are valid generic type names.

Generic functions are currently not first class functions. They can be only called, but not assigned to variables, passed as arguments or returned from functions.

Functions as types

Functions are first-class citizens in lush. They can be assigned to variables, passed as arguments or returned from functions. The type of a function is its signature. Let us consider an example from the "std:iter" module.

```
use std:iter
# In std:iter
# "filter" takes a function "filter_fn", which must return a bool and take an argument of ty
# fn filter (in: [T] ret: [T] filter_fn: fn(ret: bool arg: T))
#     ...
# end

fn is_bigger_3(ret: bool arg: num) # is_bigger_3 has such an signature
end

[1 2 3] | filter $is_bigger_3
```

As seen, writing a function-type is similar to declaring a function. Only the function name is left out.

Modules

Lush has a module system. A module is a file from which functions and struct declarations will be exported. Modules can be brought into scope via a `use` directive. There are 3 different sources of modules - Standard library modules. Those modules start with "std". (See below) - All directories under '/home//.config/lush/plugins' are assumed to be a module. - Files relative to the evaluated file.

Examples:

```
use std:array
push [] 1 2 3 # Use push from std:array
```

```
# Lets assume there is a file
# /home/<user-name>/.config/lush/plugins/my_plugin/file1.lu
# with the content:
# fn greet
#     echo "Hello from my_plugin/file1.lu"
# end
```

```
use my_plugin:file1.lu
greet         # Use greet from file1.lu

# In ./file.lu:
# fn greet
#   echo "Hi from file.lush"
# end
use ./file.lush
greet         # Use greet from ./file.lu
```

Please note: - Each evaluated file includes relative to its own path. “use ./file.lu” from “./start_file.lu” will include a different file than “use ./file.lu” from “./dir/other_file.lu”. - “use relative_file” is interpreted as a module include from “/home//.config/lush/plugins/”. Prepend a “./” to the file name to make it a relative module include. - The `use` directive, does not evaluate anything. Files imported via `use` are not run. e.G.

```
# In ./greet.lu:
# echo Hello
use ./greet.lush # Won't execute "echo Hello"
```

Debugging

`lush` offers the ability to run the code in an interactive debugger. Try `lush --debug <file>` to try it out.

The standard library

The standard library currently only consists of: - `std:array` - Exported functions - `push`: fn push(ret: [T], to_append: [T], ...elems_to_push: T) - Returns a new array which is the concatenation of `to_append` with `...elems_to_push` - `std:iter` - Exported functions - `map`: fn map (in: [T] ret: [U] map_fn: fn(ret: U arg: T)) - Applies `map_fn` to every element of `in`, collects the results in an array and returns it. - `filter`: fn filter (in: [T] ret: [T] filter_fn: fn(ret: bool arg: T)) - Applies `filter_fn` to every element in `in` and only returns those elements for which `filter_fn` returns true