JPP - language specification

Mateusz Dudziński

April 26, 2020

Changes from the previous version

- Add readonly variables and func params, as I've realized they are obligatory in the 15-points milestone.

- Don't use builtin functions - they were causing too much trouble. Use pseudo-statement print and scan function (which accept any number of args).

- Added 'assert' function that helps when testing.

- Removed 'die' function, because it is basically 'assert(false)'

- Change some operator precedence because things like: if (1 + 3 >= 4) dind't parse. Now they do.

- TODO: Something else?

The following lsanguage is an imperative, statically typed, mostly-C-like language, featuring with user-defined structs, local/unnamed functions with advanced local variables visibility options, arguments passed by value (no references), multiple return values (simple tuples) and type in variable declarations.

The 'program' in a sequence of statements written from the top to the bottom. Instructions are executed one-after-another. Functions and variables declared without parent scope are considered global. User can never redefine a global variable / function. Defined but not assigned variables don't have default value and trying to use them (like uninitialized bool in if expresion or integer for adding) causes interpreter to crash the program. Locally defined symbols are visible in their scope (like in C). User can't redefine symbol in the same scope. For example the following: `foo :  int = 0; foo :  int = 0;` would parse but fails at runtime. Same rules apply to functions and structs.

There is no `null` constant to which an expression can be compared. This is by design, becasue nulls make type deduction much harder.

Declarations:

Define variable.

```
// (type is either string / int / bool or user-defined struct (see: structs)):
variable_name : variable_type ;
foo : int ;
```

Declare and assign value.

```
foo : int = 4;
```

Declare, assign but deduce the type.

```
foo := 4;
```

The idea is that user can 'skip' the part of declaration he does not want, like skip the type if type can be deduced (deducing type works, because the language is statically typed). Also the difference between declaration and simple assigment is the : which appears in every type of declaration.

Asignments however:

```
foo = 4;
```

... don't have this token and are simple C-like assignments.

Expressions

Expressions are basic arithmetic and logic expressions taken straight from C, operator @ for string concatenation and IIFE (in grammar called EIife, described later here in the 'functions' part).

```
// Most of the syntax is C-like:
if (boolean_expr)
{
    statement();
    x := 1 + 2;
    y := "programming" @ " " @ "language";
}

if (true)
    statement();

if (ivokeFunctionThatReturnsBool())
    statement();

if (ivokeFunctionThatReturnsBool())
{
}

while (bar)
{
}

while (bar)
    if (foo)
return 5;
```

For loops

For is a little different. There is no range-for loop, fors only go from integer to integer by one. The interpreter will decide (at runtime) whether we are iterating upwards on downwards.

```
for (new_var_name : 1 .. 2)
{
}

// Of course these don't have to be constants:
for (new_var_name : begin() .. end())
{
    // for and while loops support break and continue;
    break;
    continue;
```

```
}

// Also mixed with other expressions, braces (like in C) are not needed.
for (new_var_name : begin() .. end())
    if (foo)
    {
    }

for (new_var_name : begin() .. end())
    while (foo)
    {
    }

if (foo)
    for (new_var_name : begin() .. end())
    {
    }

while (foo)
    for (new_var_name : begin() .. end())
    {
    }

{ } // Empty blocks works

;;;; // trailing ';' are accepted and not present in  ast, thanks to bnfc.
```

However things like `if ();`, `while ();` `for (...);` `else ;` won't parse. It came out a bit accidentaly, when I was trying to eliminate parsing conflicts around if/else expressions but I think it can be considered a feature.

Struct definitions. Very similar to C, just with slightly different syntax.

```
foo :: struct
{
}

bar :: struct
{
    x : int;
}

baz :: struct
{
    x : int;
    y : int;
}

v3 :: struct
{
    x : int;
    y : int;
    z : int;
}
```

```
quater :: struct
{
    e : v3;
    w : int;
}


example_vector : v3;
example_quaterion : quater;
x_copy = example_vector.x; // Getting struct members like in C.
x_copy_q = example_quaterion.e.x; // dots can follow one another.
// nope := example_vector.(e.x); // stuff like this won't parse of course.
```

Of course in the above examples, the variables would be uninitialized, which would cause a runtime error.

Structs can be defined for the scope:

```
{
    foobar :: struct
    {
zzz : string;
    }

    m : foobar;
    m.zzz = "mateusz";
}
```

Functions

Most important feature of the language is a **!** (bind) operator. This was design to make code refactoring easier by specifying which variables can be accessed in the block / lambda / function. There is a little difference between these anyway.

Super boring example, regular named (global) function.

```
exmaple1 :: (x : int, y : int) -> int
{
    return y * x;
}
```

Return type can be ommited, if function does not return.

```
out : int = 0;
exmaple2 :: (x : int, y : int)
{
    out = y * x;
}
```

Function that binds a variable - only 'foo' and function params are visible inside the function body. Everything should be an interpreter error.

```
foo : int = 12;
exmaple3 :: (x : int, y: int) !(foo) -> int
{
```

```
    return foo + y * x;
}
```

This function is pure. It is not the same as skipping '!' - single '!' means unction can refer to <u>non</u> variables (aka. is pure), skipping '!' allows it to refer to all variables (like in C).

```
foo : int = 12;
exmaple4 :: (x : int, y: int)! -> int
{
    // foo can't be accessed here, the function is pure.
    return y * x;
}
```

This function is not pure, and can reference every variable in its scope. It means global variables + local scope variables, if function is defined in the local scope.

```
foo : int = 12;
exmaple5 :: (x : int, y: int) -> int
{
    return foo + y * x;
}
```

Nested functions.

```
exmaple6 :: (x : int, y: int)! -> int
{
    square :: (x : int)! -> int
    {
return x * x;
    }

    return square(x) + square(y);
}
```

Lambda expressions. Since we don't have a higher order funcs (no passing, no returning function), all we can do with it, is to immidietly invoke it (IIFE) This is usefull when we have block that caluclates something and we want to keep it as pure as possible.

```
iife_example1 :: (x : int, y : int)!
{
    out : int = 0;

    {
x = x + 6;
y = y - x;
x = x * y;
y = x - 5;
out = x + y;
    }
}
```

We could make it a little more safe and refactoring friendly by binding x and y and out in the block, so that we can't refer to anything else.

```
foo : int = 42;
```

```
iife_example2 :: (x : int, y : int)!
{
    out : int = 0;
    !(x, y, out)
    {
x = x + 6;
y = y - x;
x = x * y;
y = x - 5;
out = x + y;
    }
}
```

We have to declare out and then change if, which is ugly and bugprone, thats where IIFE comes to help us.

```
iife_example3 :: (x : int, y : int)!
{
    // We can define out and assgin it at the same time. Assing 'out' to
    // Immidietly Called Function Expression which can refer only to x and y
    // and computes something from them as purely as it is possible.
    out : int = () !(x, y) -> int {
x = x + 6;
y = y - x;
x = x * y;
y = x - 5;
return x + y;
    }();

    // Alternatively, we could do:
    out : int = (x_ : int, y_ : int)! -> int {
x_ = x_ + 6;
y_ = y_ - x_;
x_ = x_ * y_;
y_ = x_ - 5;
return x_ + y_;
    }();
    // ... which achieves the same, but is more ugly.
}
```

The whole idea about it is that is is very easy to extract code from block into 'binded' block or iife, into local function, into global function, which all have a very similar syntax (lambda, aka. 'unnamed function' definition sytnax is the same as 'named function', but without the name). Which is not what most languages offer (like in C++, lambdas have everything differently than regular functions).

Tuples:

Tuple syntax are (exclusively) square brackets. But the amount of stuff that user can do to a tuple is very limited (by design). So there is no nested tuples, No tuple 'type' and getting a variable by name (like `foo.get<0>()` in C++) etc. The only thing user can do with a tuple is asign it or return it. However, asignment is possible with := and with =, which causes different things. := declares new variable, and = sets variables that already exists to their new values.

Since tuple is not a stand-alone expression nesting tuples or just using them as single statemetn does not parse. Using _ inside a tuple match is just an ignore. It can't however be used when <u>returning</u> tuples - in that case all values must be specified (Compare TupleExp and TupleTarget).

```
{
    // Tuples can be used to create new variables:
    [x, y] := [1, 2]; // x and y and declared here.
}


{
    // Or to assign to already existing onces (like C++'s std::tie):
    x : int;
    y : int;
    [x, y] = [1, 2];
}


{
    // Also operator '_' is supported on the lhs of the tuple assignment.
    // Noe that if trying to replace 1 or 2 with _ it would not parse,
    // bacause rhs is list of expressions, and lhs identifiers / '_'.
    [_, y] := [1, 2];
}
```

Tuples can be returned from the function:

```
tuple_example :: ()! -> [int, int] {
    x : int = 12;
    y : int = x * x;

    return [x, y];
}


// Or (of course) from the IIFE:
[x, y] := ()! -> [int, int] {
    x : int = 12;
    y : int = x * x;

    return [x, y];
}();
```

Nesting tuples is not supported. Tuple is not stand-alone expression. Empty tuples also are not supported. The following do not parse:

```
// _ = []; // as opposed to '_ = [1];' which does.
// [1];
// if ([true]) {}
```

    // Operator _ also works for assignments, but <u>not</u> for declarations, so:

```
_ = "mateusz";
_ = [ 1, 2, "mateusz" ];
_ = foobar();
// ... would parse, but:
// _ := "mateusz";
// _ := [ 1, 2, "mateusz" ];
```

```
// _ := foobar();
// ... do not.
```

The interpreter provides following 'standard library' functions:

```
read_int :: () -> [bool, int]          : Read an integer from stdin.
read_string :: () -> [bool, int]       : Read a string from stdin.
write_int :: (v : int) -> [bool]       : Write an int to stdout.
write_string :: (v : string) -> [bool] : Write a string to stdout.
die :: (msg : string)!                 : Kill program execution (with message).
```

IO functions return a boolean telling if the operation succeeded or not, read functions return a pair [`error`; `readValue`].

Disclaimer:

Most of the ideas here (especially the '!' operator, but also the basics for the assigment syntax) were invented (or at least gathered up and presented) by Jonathan Blow in his talk 'Ideas for a new programming language for games'. When he described something similar (syntax is slightly different that what I've came up with): `https://www.youtube.com/watch?v=TH9VCN6UkyQ` .

Cennik:

```
  Na 15 punktów
X 01 (trzy typy)
X 02 (literały, arytmetyka, porównania)
X 03 (zmienne, przypisanie)
X 04 (print)
X 05 (while, if)
X 06 (funkcje lub procedury, rekurencja)
X 07 (przez zmienną / przez wartość / in/out) [przez wartość]
X 08 (zmienne read-only i pętla for)
  Na 20 punktów
X 09 (przesłanianie i statyczne wiązanie)
X 10 (obsługa błędów wykonania)
X 11 (funkcje zwracające wartość)
  Na 30 punktów
X 12 (4) (statyczne typowanie)
X 13 (2) (funkcje zagnieżdżone ze statycznym wiązaniem)
X 14 (1) (rekordy/tablice/listy) [rekordy]
X 15 (2) (krotki z przypisaniem)
X 16 (1) (break, continue)
  17 (4) (funkcje wyższego rzędu, anonimowe, domknięcia)
  18 (3) (generatory)

X 19     ('bind' operator)
Razem: 30
```