JPP - language specification

Mateusz Dudziński (md394171)

May 3, 2020

Changes from the previous version

- Add 'readonly' modifier to variables 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` statement that helps when testing, with the obvious meaning and behaviour.
- Change some operator precedence because things like: {if (1 + 3 >= 4)} didn't parse. Now they do,
- Add 'new' statement for initializing structs. Since the language does not support nulls by design, when declaring variable of a struct type, there was nothing that could go to the right-hand side instead of function call, which seemed weird. The 'new' syntax is exactly the same as in C# and is presented in its 'good' test case.
- Updated documentation according to these changes.

Approach

The approach is really straight-forward.

I've decided not to use State monad, I just couldn't get some value of it and it caused some problems. Most function are built around Error monad (Error.hs), which is very similar to ErrM and I could just change that one, but I didn't want to edit machine-generated code for that. Error monad fails with `ErrorDetail` object, which describes any kind of error in the interpreter.

The program state is built on maps ("strict" versions are used for performance reasons, the program wouldn't benefit much from laziness here). State is split into two groups - Scope and Store. Scope contains a names of variables, functions and user-defined types. Store contains values. Objects are identified with ids, which are unique numbers given to them. For example, assume variable "foo" of type string and value "foobar". Scope maps "foo" into VarId = 4, and Store maps 4 into an object of type string and value "foobar". This allows for static correct binding when calling scope-defined functions (see for example test good/13 nested...). When calling a function, or any other scoped operation, we change the scope (to have old name -> object id mapping), but keep the store (to have current object values).

In order to allow for functions that return a value, break and continue statements etc, we use `CtrlFlow` functor and `CtrlT` monad (State.h). Since `CtrlFlow` is never used alone it does not have to be a monad. `CtrlT` is a monad, which combines the flow with `IO`. They wrap the Error monad also adding a `CtrlException` state. (Exception is not the greatest name, since the language don't have them, but I couldn't came up with anything better).

Many functions that don't return/break and just return Error are "promoted" to the `CtrlT` monad with the `toCtrlT` function, used across the whole project.

Static type check module is separate from the Eval module. It's very similar to Evaluation module, which makes them very similar, however I tired to reuse as many things as I could. The type-checking is basically running functions, that check correctness and then returning a "default" value of a desired type (if in the expression). The main difference is, expressions in Static module don't return a new state, because they can't change it in any meaningful way for the type checker (evaluating a single expression, i.eg.: a function argument), can't create a new type or change a type of any variable visible in the scope.

Makefile and running the test suite

`make` command will build the interpreter for sources. It won't by default regenerate BNFC files. For that use `make language`. It will start BNFC suite programs and then copy generated files to their appropriate directories, removing ones we don't want. All tests are in the `test` directory. There is a script to run the test suite, but easiest way to do it is by typing `make validate` into the command line. Tests that are supposed to fail have `//FAILS:` comment at the top. It is used for output comparison when testing. It is possible to disable static type checking in the `Main.hs` file and all tests should produce exactly the same results, including the error messages, expect all 12* tests, which should all fail in the first line because of the broken assertion.

Running the program

Interpreter can either take one and only parameter, which is a path to a file to execute, or when no params it reads the text from stdin. Note that when reading stdin, `scan` function will never work, and will always return value that indicates an error. Because every line starting with # is ignored by the parser one can put

```
#!/path/to/interpreter/can/be/relative
```

at the first line of the file and run the files directly from the shell.

Basics:

The following language 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 expression- 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, because nulls make type deduction much harder. Each type declared but not given an initial value will have the value equal to the default of its respective type. It means: `0`, `""` and `false` for `int`, `string` and `bool`. Structs have all their members initiated recursively.

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.

Also read-only variables are supported:

```
foo! := 4; // Foo is now read-only.
foo = 4; // Would cause an error.
```

Expressions

Expressions are basic arithmetic and logic expressions taken straight from C, expect 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, for loops only go from integer to integer by one. The interpreter will decide (at runtime) whether we are iterating upwards on downwards. Iterator variable is read-only in the loop body, so can't be reassigned.

```
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 accidentally, 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";

    // 'new' syntax allowes us to assigne sturct fields on
    // declaration, avoiding anti-patter above. This is the same:
    n := new foobar { zzz = "mateusz" };

    // Of course type can be also given explicitely, but in this case
    // it is rather pointless.
    n' : foobar = new foobar { 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.

```
// Function parameters also support read-only attribute. X can't be changed inside func body.
exmaple1 :: (x! : int, y : int) -> int
{
    return y * x;
}
```

Return type can be omitted, 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 non 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 syntax 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 assign it or return it. However, assignment 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 statement does not parse. Using `_` inside a tuple match is just an ignore. It can't however be used when returning 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 not for declarations, so:

```
_ = "mateusz";
_ = [ 1, 2, "mateusz" ];
_ = foobar();
// ... would parse, but:
// _ := "mateusz";
// _ := [ 1, 2, "mateusz" ];
// _ := foobar();
// ... do not.
```

The interpreter also provides `print`, `scan` and `assert` expressions

- `print` - prints the expression to the screen. Must be a builtin type. Printing structs is not supported, printing tuples would not even parse, because tuples are not expressions.
- `scan` - scan is a pseudo-statement that returns $n+1$ - element tuple, where n was a number of its parameters and tries to scanf these from the next stdin line. Whole line is fetched, so each scan must expect a separate, one newline. Also only builtin types are expected. The first element of the tuple tells the user how many elements were scanned properly, 0 means none. The rest of the unscanned args have their default values (since null is not supported). The function does not distinguish from IO error and parasing error and will return 0 as first argument on any IO error.
- `assert` - obvious. Used mostly in testing.

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 99     ('new' syntax)
X 99     (iife)
X 99     ('bind' operator)

Razem: 30
```