The nk hardtyped language

May, 2022

Project Team

Maxim Ksenofontov	Type checker
Vladislav Lamzenkov	Interpreter developer
Dmitrii Polushin	Tests, documentation
Vladislav Levchenko	Language design, BNF

Ideas of the language

Our language syntax and semantics are going to be similar to functional language. It should be used as a first language to study functional programming. At the same time, it should be a typed language, so students are able to infer type by themselves. It would be verified by the type-checker that is the part of the interpreter. Because of our language is functional every expression should return something, at least *Unit* type. This is affects functions and what they are returns, but we will discuss it later. The syntax of our language is not obvious at some moments, but we inspired by lambda calculus in our design and hope that future developers will encourage with this interesting solutions.
Link to Github: Link

List of Features

Here is the list of main features of our language:

Type Checker
Base Types (integer, real, booleans, strings, unit)
User-defined terms and types (structures)
Standard library (e.g. arithmetic, logical, lists)
First-class functions (anonymous functions)
Nested definitions (nested functions)
Simple Constraint-Based Type Inference (e.g. support auto types)
Functions with multiple (but fixed amount) arguments
General Recursion
Simple modules and imports
Records
Subtyping

The lexical structure of hardtyped

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters `_ '`, reserved words excluded.

Literals

Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression ⟨*digit*⟩ + '*.*'⟨*digit*⟩ + ('*e*''*-*'?⟨*digit*⟩+)? i.e. two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent.

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

String literals ⟨*String*⟩ have the form `"`*x*`"`, where *x* is any sequence of any characters except `"` unless preceded by `\`.

Bool literals are recognized by the regular expression {"true"} | {"false"}

Unit literals are recognized by the regular expression {"unit"}

Print literals are recognized by the regular expression {"print"}

ReadReal literals are recognized by the regular expression {"readReal"}

ReadInt literals are recognized by the regular expression {"readInt"}

ReadString literals are recognized by the regular expression {"readString"}

ReadBool literals are recognized by the regular expression {"readBool"}

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in hardtyped are the following:

`Any`	`Bool`	`Int`
`Real`	`String`	`Unit`
`and`	`as`	`in`
`let`	`letrec`	`lettype`
`not`	`or`	

The symbols used in hardtyped are the following:

;	(	)
-|	=	:
/\	{	}
->	.	,
|	|:	+
-	*	/
>	>=	==
!=	<=	<

Comments

Single-line comments begin with //.
Multiple-line comments are enclosed with /* and */.

The syntactic structure of hardtyped

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

$$\begin{array}{lll} \langle \mathit{ListExpr} \rangle & ::= & \langle \mathit{Expr} \rangle \ \texttt{;} \\ & | & \langle \mathit{Expr} \rangle \ \texttt{;} \ \langle \mathit{ListExpr} \rangle \end{array}$$

```
⟨Expr⟩  ::=  ⟨Expr1⟩
          |  −| ⟨String⟩ as ⟨Ident⟩
          |  −| ⟨String⟩
          |  let ⟨VarDec⟩ = ⟨Expr1⟩
          |  let ⟨VarDec⟩ = ⟨Expr1⟩ in ⟨Expr1⟩
          |  letrec ⟨VarDec⟩ = ⟨Expr1⟩
          |  letrec ⟨VarDec⟩ = ⟨Expr1⟩ in ⟨Expr1⟩
          |  lettype ⟨VarDec⟩ = ⟨Type⟩
          |  lettype ⟨VarDec⟩ = ⟨Type⟩ in ⟨Expr1⟩
          |  ⟨ListIfExpr⟩ ⟨ElseExpr⟩

⟨Expr1⟩  ::=  ⟨Expr2⟩
           |  /\ ⟨ListFuncArg⟩ { ⟨ListExpr⟩ }
           |  ⟨Print⟩ ( ⟨Expr⟩ )
           |  ⟨ReadReal⟩ ( ⟨ListExprSequence⟩ )
           |  ⟨ReadInt⟩ ( ⟨ListExprSequence⟩ )
           |  ⟨ReadString⟩ ( ⟨ListExprSequence⟩ )
           |  ⟨ReadBool⟩ ( ⟨ListExprSequence⟩ )

⟨Expr2⟩  ::=  ⟨Expr3⟩
           |  /\ ⟨ListFuncArg⟩ { ⟨ListExpr⟩ } ( ⟨ListExprSequence⟩ )
           |  /\ ⟨ListFuncArg⟩ { ⟨ListExpr⟩ } −> ⟨Type⟩
           |  ⟨Op⟩

⟨Expr3⟩  ::=  ⟨Expr4⟩
           |  ⟨Expr4⟩ ( ⟨ListExprSequence⟩ )
           |  ⟨Integer⟩
           |  ⟨Double⟩
           |  ⟨String⟩
           |  ⟨Bool⟩
           |  ⟨Unit⟩
           |  { ⟨ListRecordElem⟩ }

⟨Expr4⟩  ::=  ⟨Expr5⟩
           |  ⟨Ident⟩
           |  ⟨Ident⟩ −> ⟨Expr4⟩
           |  ⟨Expr4⟩ . ⟨Expr4⟩

⟨Expr5⟩  ::=  ( ⟨Expr⟩ )
           |  ( ⟨ListExpr⟩ )

⟨VarDec⟩  ::=  ⟨Ident⟩ : ⟨Type⟩
            |  ⟨Ident⟩

⟨ExprSequence⟩  ::=  ⟨Expr⟩
```

$$\begin{array}{lll}
\langle \textit{ListExprSequence} \rangle & ::= & \epsilon \\
 & | & \langle \textit{ExprSequence} \rangle \\
 & | & \langle \textit{ExprSequence} \rangle \ \texttt{,} \ \langle \textit{ListExprSequence} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{FuncArg} \rangle & ::= & \langle \textit{Ident} \rangle \ \texttt{:} \ \langle \textit{Type} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{ListFuncArg} \rangle & ::= & \langle \textit{FuncArg} \rangle \ \texttt{.} \\
 & | & \langle \textit{FuncArg} \rangle \ \texttt{.} \ \langle \textit{ListFuncArg} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{IfExpr} \rangle & ::= & \texttt{|} \ \texttt{(} \ \langle \textit{Expr2} \rangle \ \texttt{)} \ \texttt{:} \ \langle \textit{Expr} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{ListIfExpr} \rangle & ::= & \epsilon \\
 & | & \langle \textit{IfExpr} \rangle \ \langle \textit{ListIfExpr} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{ElseExpr} \rangle & ::= & \texttt{|:} \ \langle \textit{Expr} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op} \rangle & ::= & \langle \textit{Op1} \rangle \\
 & | & \langle \textit{Expr3} \rangle \ \texttt{or} \ \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op1} \rangle & ::= & \langle \textit{Op2} \rangle \\
 & | & \langle \textit{Expr3} \rangle \ \texttt{and} \ \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op2} \rangle & ::= & \langle \textit{Op3} \rangle \\
 & | & \texttt{not} \ \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op3} \rangle & ::= & \langle \textit{Op4} \rangle \\
 & | & \langle \textit{Expr3} \rangle > \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle >= \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle == \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle \ \texttt{!=} \ \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle <= \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle < \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op4} \rangle & ::= & \langle \textit{Op5} \rangle \\
 & | & \langle \textit{Expr3} \rangle + \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle - \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op5} \rangle & ::= & \langle \textit{Op6} \rangle \\
 & | & \langle \textit{Expr3} \rangle \ \texttt{*} \ \langle \textit{Expr3} \rangle \\
 & | & \langle \textit{Expr3} \rangle \ \texttt{/} \ \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{Op6} \rangle & ::= & \texttt{(} \ \langle \textit{Op} \rangle \ \texttt{)} \\
 & | & + \ \langle \textit{Expr3} \rangle \\
 & | & - \ \langle \textit{Expr3} \rangle
\end{array}$$

$$\begin{array}{lll}
\langle \textit{RecordElem} \rangle & ::= & \langle \textit{Ident} \rangle = \langle \textit{Expr3} \rangle
\end{array}$$

```
⟨ListRecordElem⟩  ::=  ϵ
                    |  ⟨RecordElem⟩
                    |  ⟨RecordElem⟩ , ⟨ListRecordElem⟩

⟨Type⟩  ::=  ⟨Type1⟩

⟨Type1⟩  ::=  ⟨Type2⟩
          |   ⟨Type1⟩ −> ⟨Type2⟩

⟨Type2⟩  ::=  ⟨Type3⟩
          |   ⟨Ident⟩
          |   Int
          |   Real
          |   Bool
          |   String
          |   Unit
          |   Any
          |   { ⟨ListRecordElemType⟩ }

⟨Type3⟩  ::=  ( ⟨Type⟩ )

⟨RecordElemType⟩  ::=  ⟨Ident⟩ : ⟨Type⟩

⟨ListRecordElemType⟩  ::=  ϵ
                        |  ⟨RecordElemType⟩
                        |  ⟨RecordElemType⟩ , ⟨ListRecordElemType⟩
```

Variable Declaration

Primitive types

Any - parent type of all types
Int - integer numbers(4 bytes)
Real - real numbers(8 bytes)
Boolean - true / false
String - raw text
Unit - empty

User types

In nk language we have no exact user defined types, but we have type aliases that are used for grouping logically related types into a single construct.

Example:

```
// Alias for record with 3 keys x1, x2 and x3
lettype Vector3d = {x1: Real, x2: Real, x3: Real};

// Alias for basic type real
lettype Money = Real;

// Function that takes Int as an argument and returns string as
    a result
lettype IntToStringFunc = (Int -> String);
```

Variable declarations

Example:

```
-| "std";

let n: Int = 5;
let r: Real = 10.5;
let b: Boolean = true;
let s: String = "Simple string";
let x: Int = 10 in 100 + x;
let u: Unit = print (s);
```

In our language we have local and global variable declarations.

For local declarations use *let in* construction. In this case, the variable will be accessible only inside *in* expression.

Example:

```
-| "math";

let x: Int = 10 in 100 + x;
sqrt(x); // Error: x is not declared in the scope.
```

For global declarations use *let*. In this case the variable will be available everywhere.If we define global variable in one program file, we can use it inside other modules using import system.

```
-| "math";

let x: Real = 10.0;
sqrt(x); // No error
```

However, you cannot use *let* construction in local scopes. You may declare global variables only being in global scope:

```
let identity = /\ x: Any. {
    // Error: You cannot declare global variable 'itWillFail'
    in local scope
    let itWillFail = "Hello...";
};

identity(5);
```

Besides, we support variable shadowing: in local scopes you may define variables with the same name as in the upper scopes. In this case, the variables with the same names in scopes above won't be accessible.

Example:

```
 // The following program outputs:
 // 10
 // 5
-| "std";

let x = 5 in (
    let x = 10 in print (x);
    print (x);
);
```

Type inference

Also, our language support type inference for declarations. It is not necessary to exactly write the type of variable, as a result, it will be expressed from assigned expression. This check is performed by our type checker before interpretation stage.

Example:

```
-| "std";

// Type of x is Int
let x = 5 + 1;

// Type of y is Real
let y = 5.0 + intToReal(2);
```

Expressions

Relational

Relational operations that supported in our language.

- **Comparisons**. Supported operands of comparisons: <, >, <=, >=, ==, !=. To compare different types the explicit cast is required.

  The principle of comparisons:

  ```
      Int op Int -> Boolean
      Real op Real -> Boolean
      String op String -> Boolean
  ```

- **Logical operations**. We have three basic logical operands: and, or, not. The results of logical operations:

```
    Boolean and Boolean -> Boolean
    Boolean or Boolean -> Boolean
    not Boolean -> Boolean
```

Other types of operands are not allowed.

Arithmetic

Arithmetic operations that supported in nk language:

- Addition (+)

```
    Int + Int -> Int
    Real + Real -> Real
    String + String -> String (string concatenation)
```

- Substraction (-)

```
    Int - Int -> Int
    Real - Real -> Real
```

- Multiplication (*)

```
    Int * Int -> Int
    Real * Real -> Real
```

- Division (/)

```
    Int / Int -> Int (round down)
    Real / Real -> Real
```

Example:

```
let x = 5 + 5; // 10
let y = (true and false) or true; // true
let z = (2 + 2) * 2; // 8
let k = (10 >= 6) and (2 == 3); // false
```

Because our language is hardtyped you can not perform operations between different types. To operate with different types you need to explicitly cast on of this type into another:

```
-| "std";

let y = realToInt (5.0) + 2; // type of y will be Integer
```

You should directly define the order of execution with brackets for complex calculation with multiple steps:

```
let b = (((4 + (5+1))*3)*4)/2;
```

Modules and Imports

You can import any modules in any scope, all global definitions from the specified file will be available automatically inside your program file. For it just use construction -| *"modulename"*.

Example:

```
-| "std";

let f = /\ x: Real. {
    -| "math";
    print(sqrt(x));
};

f(4.0); // Outputs: 2.0
// However, if you try to call sqrt() here, the error will be
   raised.
// sqrt(4.0);
```

For now, you may only import libraries(*std, math*), but we have an interesting thing. Our modules may be written using Java language via following the contract defined by our API. To know more about it, please, visit the package *com.interpreter.shared.libs* in the source files of the project.

Records

The syntax of defining a record is the following via brackets . If we want to access specific element in record we call it as method via dot *"."*.

Example:

```
-| "std";

lettype Vector3d = {x1: Real, x2: Real, x3: Real};
// Defining record
let v: Vector3d = {x1=1.1, x2=2.2, x3=3.3};
let r: Real = v.x1;

// Output:
// 3.3
print (v.x3);
```

Also, the records can be nested inside of each other. In this case, to access them use nested dots *"."*.

Example:

```
-| "std";

lettype NestedRecord = {x1:{a:Int, b:Int}, x2:Int, x3:Int};
let r: NestedRecord = {x1 = {a = 1, b = 2} , x2 = 2 , x3 = 3};

```

```
print(r.x1.a); // Output: 1
```

Functions

The simplest function in our language:

```
let id = /\ x:Any.{
    x;
};
```

Function contains a set of expressions that work with arguments that you passed inside it. Besides, we do not have explicit return statement, but the last expression is returned as result of function execution. In addition, we designed in the following way that every function must have at least 1 argument, and when you call a function using parenthesis you must pass at least 1 argument. Otherwise, the error will be raised.

Example:

```
-| "std";

// Returned type: Unit
/\ x:Int.{
    x+x;
    print(x); // result of this expr is returned
};

// Returned type: Int
/\ x:Int.{
    print(x);
    x+x;
};
```

We can explicitly define the type that our function returns, and type checker will check that returned type of last expression is the same (or a subtype of it).

Example:

```
-| "std";

/\ x:String. y:String. {
    x+y;
} -> String;

// Type check error
/\ x:String. y:String. {
    x+y;
    print (x+y);
} -> String;
```

Multiple arguments of the function are separated with dots as in lambda syntax.

Example:

```
/\ x:Int. y:Int. {
    x + y;
};
```

In addition, when you define a function, its outer context(all scopes) is captured. When you invoke it, then this context will be used. It is done for state-safe programming approach to disallow developers to see unexpected behaviour.

Correct example:

```
-| "std";

let m = 5;
let f = /\ x: Unit. {
    print(m);
};

f(unit); // Output: 5
```

Incorrect example:

```
-| "std";

let f = /\ x: Unit. {
    print(m);
};

let m = 5;
f(unit); // Error: m is not declared in the scope
```

Types of supported functions:

- Nested functions
  This is the function that is defined within another function and can be used only inside of inner function. If you want to define nested function you should use let in construction, because you need to define this function in local scope. The example of a nested function:

  ```
      -| "std";

      // Output:
      // S
      // S
      /\ x: String. {
          let nested_func = /\ y: String. {
             print (y);
          } in nested_func (x);
          print (x);
      } ("S");
  ```

- First-class functions
  You may pass function as argument to another function, return them as the values from other functions, and assign them to variables.

```
-| "std";

let first_class_function = /\ x: String. {
    /\ y: String. {
       print (x + y);
    };
};
let result_function = first_class_function ("Hello, ");
result_function ("World!");
// Output:
// "Hello, World!"

```

- Recursive Functions
  For this kind of function we need to use special construction *letrec*, instead of just *let*. Another significant moment of recursive functions is that it is mandatory to define the returned type of such function, since we do not support recursive types. In addition, an attempt to use of *let* construction for the recursion, you will get error showing your function is not declared in the scope. It happens also because of context capture.

  Example:

```
-| "std";

letrec factorial =
/\ n: Int. {
    |  (n == 1): n
    |:  n * factorial (n-1);
} -> Int;

print(factorial(3)); // Output: 6

```

Subtyping

The following function accepts argument that has type *Any*. Then we can pass inside it all types that inherited from it. It is important to note that all types are inherited from *Any* by default.

Example of function with subtyping:

```
-| "std";

let id = /\ a: Any. {
    a;
};
print(id (5));// Output: 5
```

As argument of function we are expecting record Vector3d with three elements inside. But if we pass expanded version of this record with extra parameter x4, it still works because of subtyping. The main thing in record subtyping is that we can only expand records not changing the original record.

Example:

```
-| "std";
-| "math";

lettype Vector3d = {x1: Real, x2: Real, x3: Real};

let length3d = /\ v: Vector3d. {
    // Output: 2.5605261936270534
    print(sqrt (sqrt (v.x1) + (sqrt (v.x2) + sqrt (v.x3))));
} in (length3d ({x1 = 1.1, x2 = 5.5, x3 = 10.0, x4 = 1.7}));
```

Function subtyping

The basic rule for function subtyping is that argument types must be contravariant, but return types must be co-variant

Correct example:

```
// In this example, func2 is subtype of func1
lettype func1 = {x1:Int, x2:Int, x3:Int}->{x1:Int, x2:Int};

lettype func2 = {x1:Int, x2:Int}->{x1:Int, x2:Int, x3:Int};
```

Incorrect example:

```
// In this example, argument types are not contra-variant and
    returned type of func2 is not expanded, but changed from
    type return in func1.
lettype func1 = Int->{x1:Int, x2:Int};

lettype func2 = {x1:Int, x2:Int}->{x2:Int, x3:Int};
```

Conditions

Conditions in our language can be inside and outside of functions. In other words, they are just expressions. There is no difference between *if* and *elif* statements in the syntax of the language. In addition, please, note that ";" separator should be only after last condition (else) and should not be at *if* and *elif* conditions.

Example:

```
/\ n: Int. {
    | ((n > 1) and (n < 5)): (n + 5) * 10
    | ((n >= 10) or (n < 0)): n * (3 + 5)
    |:  n * (n+1);
};
```

It is important to have a supertype for all the return types in all branches. If we want to execute multiple expressions inside of one condition, they must be combined using parenthesis *()*. In this case, the returned type from condition will be the type of last expression in the parenthesis.

Example:

```
-| "std";

let cond = /\ n: Int. {
    |  (n > 1): (
        print (n+10);
        n*n;
        )
    |: n-1;
};

// Output:
// 12
// 4
print (cond (2));
```

Standard Library("std")

Input and output functions are part of the standard library(*std*). This library is written on Java following our shared API package and loaded into runtime via import manager.

We have the following functions that are available in *std*:

- readReal(unit) - read real number from the user's console;
- readString(unit) - read string from the user's console;
- readBool(unit) - read bool value from the user's console;
- readInt(unit) - read integer number from the user's console;
- print(any) - prints to the console passed value;
- intToReal(int) - convert integer to real
- intToString(int) - convert integer to string
- realToInt(real) - convert real to int

- realToString(real) - convert real to string
- stringToReal(string) - convert string to real
- stringToInt(string) - convert string to int

Example(simple I/O program):

```
-| "std";

print ("Input your name");

let name = readString(unit);
print ("Hello, " + name);
```

Type Checker

All expressions, functions, and declarations are checked with the type checker before they are interpreted. The type checker works statically without executing of any expression. Besides, in our language as it is hard-typed you are not allowed to have ill-typed expressions.

Type checker checks that the return types of all the branches have a supertype, which is one of them. Type checker is able to infer it.

Correct Example:

```
/\u:Unit. {
    | (true): {x1=1.0, x2=0.0}
    | (false): {x1=1.0, x2=0.0, x3=3.0}
    |: {x1=0.0};
}; // Return type is {x1:Real}
```

Incorrect Example:

```
lettype Vec2d = {x1: Real, x2: Real};

/\u:Unit. {
    | (true): {x1=1.0, x2=0.0}
    | (false): {x1=1.0, x2=0.0, x3=3.0}
    |: {x1=0.0};
} -> Vec2d; // Specified type is different from the actual one
```

Usually, when branches do not have a common supertype, it is considered as an error. Since it makes little sense to return type Any or some other supertype. The following example produces an error, because in the first condition branch the return type will be Unit, and in other branches the returned type will be Int, as a result, they do not have a common supertype:

```
-| "std";

```

```
// Error
/\ n: Int. {
    | ((n > 1) and (n < 5)): (
        let x = (n + 5) * 10 in
            x*x;
        print(n); // returns unit
    )
    | ((n >= 10) or (n < 0)): (n * 3) + 5 // returns int
    |:  n * (n+1); //returns int
};
```

Another example, it will produce an error, because in first condition branch the return type is Unit, however, in other branches the returned type is Int. Also, type checker check the declarations:

```
let x: Int = 5.2; // Error in type checker
let y: String = 10;  // Error in type checker
let z: Bool = "String";  // Error in type checker
```

Project Usage Instructions

Build the project

Project pre-requirements:

- JDK - 17 (language level 17)
- Maven - 3.6.3
- Bnfc - 2.9.4
- Latexmk - 4.77

The steps to run the project

1. Clone the project from Git Repo.
2. Navigate inside the folder with the project
3. Run in your terminal:
   build_project.sh
4. Create a file with your extension *".nk"*(e.g. test.nk), and write your code there.
5. Build the project:
   mvn clean install
6. Run your program in the terminal(passing the last argument as path to your file with program):
   java -jar ./target/lambda-interpreter-1.0-SNAPSHOT-jar-with-dependencies.jar ./test.nk

Working programs

Working cases in our language are divided logically by grouping of features in */src/test/resources/*. There are examples of correct and incorrect programs. Be careful, incorrect programs are marked with "..._incorr.nk" naming.

To run all this programs or just some specific case you can go into */src/test/-java/*. This folder has the same structure as the resource folder. You can choose a class with specific feature and run it, or run all classes in this java folder.

Also, you may have to take a look at folder "/examples" to see some sample programs!