LINGI2132 - Languages and Translators
Building our own language
Part 3&4 - Interpreter and Final Submission

de Voghel Brieuc - 5910 1600
Kneip Elisabeth - 2106 1700

Friday 14th May, 2021

Abstract

This document summarises the features of our Kneghel[1] programming language. We go through the different features Kneghel offers, clarifying what they are and explaining their semantics.

Since last submissions, the grammar and semantics have been revisited and thoroughly corrected to be more consistent and well-founded. This has allowed for an appropriate implementation of the interpreter. We consider this version to compensate for the shortcomings of the previous submission.

1 Required features

Our language is dynamically typed, but resembles Java and its OOP style: the functions are written in classes where the `main` function is the entry point for the execution of a program. This does not directly imply that Kneghel is Object Oriented (yet).

1.1 Literals

The literals are quite straightforward. We have five different types : integers, floats, strings, booleans and null. Strings use double quotation marks around them to mark their type. Integers are stored using Java's Long type, floats using the `Double` type.

```
1       // an integer
2.0     // a float
"Hello" // a string
true    // a boolean
null    // a null instance
```

1.2 Comments

As seen in the previous code snippet, line comments are marked with a double slash. We also have block comments. Those are inspired by Java and are represented between '/*' and '*/', separated by as many lines as the user wishes.

Comments are considered to be equivalent to white spaces when parsed and are thus ignored.

```
// This is a line comment
/*
 This is a block comment
 */
```

1.3 Variable definitions

Variable definitions are done in Python style : just use the variable name followed by '=' and its value.

As said before, our language is dynamically typed. Indeed, when defining a variable, we do not need to indicate its type. The dynamic typing allows us to overwrite variables.

When declaring a new variable, its semantics are inferred from the value it is being assigned. When there is uncertainty or when the type can simply not be determined, it is assigned the `UNKNOWN_TYPE` type. This

[1] The name Kneghel originates from the contraction of Kneip and Voghel.

implies that the semantics of Kneghel are very permissive, allowing for more control to the user, but may result in more runtime errors when conflicts happen.

The scope of the variables are always limited to the block of braces they are first declared in. They can nonetheless be modified in subsequent scopes (they will then refer to the parent scope).

```
x = "this is a string"
x = 11 // x is now an integer ...
x = true // ... and now a boolean
```

1.4 Simple arithmetic

The arithmetic operations look like most other languages. The following operations are implemented : *, /, %, +, -; in this order of precedence. When prefixes (-, !) are used, they have precedence. Parenthesis are also available.

It is worth to mention that the division by zero is handled differently depending on the type of the operands. On one hand, if the division imply floats, the result is (positive or negative) infinity. On the other hand, if the division only implies integers, the operation throws an `ArithmeticException`.

Kneghel does not yet support arithmetic operations on Strings.

```
x = 1 + 2                          // x = 3
x = 2 * (4-1) * 4.0 / 6 % (2+1) // x = 1.0
x = 5 / 0.0                        // x = +infinity
x = 5 / 0                          // throws ArithmeticException
```

1.5 Conditions

Kneghel's conditional constructs looks unsurprisingly a lot like Java's, without the need of parenthesis, but with the need of braces. It starts with the keyword `if` followed by a condition. Then comes the true statement, possibly followed by the false statement, both contained in between braces. A condition needs to be of the boolean type.

The conditional operators are identical to Java's : && for `AND` and || for `OR`. Their precedence is also implemented such that an `AND` operator is applied before an `OR` operator. All operands are not necessarily evaluated, as do most other languages. An example of not all operands of a conditional statement being executed can be found in InterpreterTests.testConditionEvaluation().

Kneghel has six inequality operators (==, !=, <=, <, >=, >). They are used to compare two expressions and they return a boolean.

```
x = null
if false || 1 + 2 * 3 <= 4 && true {
    x = "should not be this"
}
else if 2 + 2 != 4 {
    x = "should not be this"
}
else if 2 + 2 == 4 {
    x = "should be this"
}
else {
    x = "should not be this"
}
// the variable x should be equal to "should be this"
```

1.6 Loops

Kneghel has only one type of loop, the `while` loop. The condition of a while loop is identical to the one of a conditional construct. Again, the statements of the loop are contained in between braces. As the example shows, and as discussed in Section 1.3 Variable definitions, the scope is popped after a while statement. The variable 'c' is not reachable outside the loop, but both 'a' and 'b' have been modified.

```
a = 1
b = 0
```

```
while false || 1 + 2 * 3 <= 4 && true || a <= 3 {
    c = b
    b = c + a
    a = a + 1
}
d = c // throws an AssertionError
/* the variable 'a' should be equal to 4 and the variable 'b' should be equal to 6.
The variable 'c' is not reachable anymore */
```

1.7 Function definition

Function signatures start with the keyword `fun` followed by the name of the function and then parentheses with optionally the arguments separated by a comma. The body of the function is then found in between braces. The return statement is optional, i.e. functions without a return statement return null, like in Python. The types of the arguments and the return type do not need to be specified anywhere.

Functions can be declared inside another (nested functions). Those ones are then only reachable in the function they are declared in.

```
fun new_fun (arg1, arg2) {
    x = "this is a string"
    print(x)
    return_value = arg1 + arg2
    return return_value
}
```

1.8 Array and map access

Arrays are abstract structures that may contain different types. The user is allowed to combine different literals and other structures within it (e.g. array of arrays).

A particularity of Kneghel is that the arrays do not have fixed sizes. They can be extended at will. Indeed, when creating an array, the user calls the built-in function `makeArray()` that does not need a size as a parameter. When an element has to be added to the array, a simple assignment suffice. The right side of the assignment is the array access at the wanted index, and the left side is the value to be added. If the index is larger than the size of the array, Kneghel will automatically fill all the missing elements with a null instance.

The map data structure is a dictionary. They are implemented with Java's `HashMap`. Every action on a dictionary is done with built-in functions. To create one, the function `makeDict()` has to be called. To add a value, you call the function `dictAdd(dict, key, value)` which is not an in-place function, i.e. it returns a new dictionary without modifying the original one. Finally, to get a value with a specified key, the user calls the function `dictGet(dict, key)`. This last function returns the value corresponding to the key key in the dictionary `dict`.

The decision of making dictionaries not in-place comes from the fact that we do not want to implement a remove function. This also makes the user able to easily track changes within dictionaries by assigning the modified ones to new variables.

```
class new_class {
    fun main () {
        new_dico = makeDict()
        new_array = makeArray()

        newest_dico = dictAdd(new_dico, "here", "a value")
        new_array[5] = "a value" // adds the string "a value" at the index 5

        return new_array[5] == dictGet(newest_dico, "here")
    }
}
```

1.9 Printing to std out

To make printing more pleasant, Kneghel's printing function was inspired by Python's. It is a built-in function callable with `print()` with an argument being what the user wants to print. The built-in function uses Java's

`System.out.println()` which allows Kneghel to print any literal and more complex data structures with a line carry. It is not yet possible to print without a line carry.

In the examples below, we also showcase the use of arguments (clarified in Section 1.10) and the use of built-in functions like `len()` and `int()`, which have the same behaviour as Python's.

```
/* args = new ArrayList<String>(){{
                    add("0");
                    add("1");
                }}
   for more detail on args, see Section 1.10 */
class new_class {
    fun main () {
        first_argument = int(args[0])
        print(first_argument) // prints 0\n
        second_argument = int(args[1])
        print(second_argument) // prints 1\n
        number_of_arguments = len(args)
        print(number_of_arguments) // prints 2\n
    }
}
```

1.10 Passing string arguments to the program

Kneghel's way of passing string arguments to the program resemble Java's way. It uses the keyword `args` which is a reference to an array of strings. Unlike Java, it is passed implicitly as an argument to the `main` function.

When instantiating the Java interpreter for a Kneghel code, the constructor expects to receive an `ArrayList` of strings as its second argument. This way, arguments are passed to the program. This is showcased in test.java.examples.KneghelExamplesInterpreterTests.

```
class new_class {
    fun main () {
        first_argument = int(args[0])
        second_argument = int(args[1])
        number_of_arguments = len(args)
        return first_argument + second_argument + number_of_arguments
    }
}
```

1.11 Parsing strings representing integers

To parse the incoming strings from `args` - but also to parse any other string - into an integer, Kneghel has a built-in function `int(string)` that takes a string as argument and return the integer represented by the string.

If the string does not represent an integer, a new NumberFormatException will be thrown.

```
class new_class {
    fun main () {
        str = "42"
        fortytwo = int(str)
        print(str)      // prints 42\n
        print(fortytwo) // prints 42\n
        not_an_int = int("this is not an int") // throws a NumberFormatException
    }
}
```

2 Extra supported features

Following features were added in addition to the required features. We implemented them to put forward the characteristics and specificities of Kneghel.

2.1 Universal access to args

We modified Kneghel so that any function can have access to the agrs parameter, passed at the execution of a Kneghel program. In this way, there is no need to complicate function definitions in order to allow a user to use its passed arguments.

This feature is showcased in the Kitchensink program (`src/examples/Kitchensink.kneghel`).

2.2 Built-in function : random int generator

To showcase the ease of implementing additional built-in functions, the `randInt` function has been implemented. It is an easy and straightforward implementation. The following Git diff shows what parts of the code have to be modified to allow for a new built-in function implementation. Of course, let's not forget to implement unit tests for the implemented functions.

The `randInt` function accepts 2 parameters which define the (inclusive) range of the returned int.

```
// in the public variables of scopes.RootScope
(+) public final SyntheticDeclarationNode _randInt = decl("randInt", FUNCTION);

// in the constructor of scopes.RootScope
(+) reactor.set(_randInt, "type", Type.INTEGER);

// extend the case statement of the builtin() function in interpreter.Interpreter
(+) case "randInt":
(+)     if (args.length != 2)
(+)         throw new PassthroughException(new IllegalArgumentException("(...)"));
(+)     long min = (long) args[0];
(+)     long max = (long) args[1];
(+)     return min + (long) (Math.random() * (max - min + 1));
```

When defining other built-in functions in the constructor of `RootScope`, it is important to change `INTEGER` according to the return type of the function (possibly `UNKNOWN_TYPE`).

3 Observations

We have not yet been able to implement support for arithmetic operations on Strings.

Other interesting observations have been discussed in the previous sections where adequate (e.g. the fact that functions may be nested in others) and do not need further explanation here.