LINGI2132 - Languages and Translators
Final Project

d'Oultremont Augustin - 2239 1700
Lemaire Valentin - 1634 1700

Group AC

Wednesday 12th May, 2021

Introduction

This report is the final report of the LINGI2132 - Languages & Translators course given by Pr. Nicolas Laurent at UCLouvain. It describes each feature that has been implemented, along with its semantic, starting with the minimal requirements and continuing on with the extra feature.

1 Minimal requirements

1.1 Literals

We have implemented integers, strings and booleans in our language. Strings start and end with double quotes and booleans are either `True` or `False`. Integers are stored as Longs in java, using 64 bits to store the value.

Examples

```
string = "PO-TA-TOES... Boil'em, mash'em, stick'em in a stew"
integer = 123
boolean = True
```

1.2 Comments

The comments have been implemented with the # character. The comments are ignored by the parser and there is thus no semantic for this part.

Examples

```
a = 1 # Setting a to 1
# One line comment
```

1.3 Variable definitions

Our language has no explicit typing. Therefore variable definitions do not have typing and are a simple assignment. Because of this there is no difference between declaration and initialization of new variables, i.e. if there is a variable assignment and the variable is already in scope, it is overwritten, otherwise it is declared and initialized. This means that there is no need for default values for variables (except for the contents of arrays, which is discussed later). Since there is no typing, a same variable can be assigned to values of different types without restriction.

Examples

```
a = 1
a = "test"
a = True
```

1.4 Simple arithmetic

The arithmetic of our language consists of the 5 main operators (+, -, *, /, %). These operations can only be performed on integers and will be executed in the usual order (first *, / and %, then + and -, there is a small difference to classic languages concerning negation, this is discussed in section 3). Parenthesis have priority over the rest of operations.

Examples

```
a = 1 + (-2) * 3
b = a / 2 * (2 + 2)
c = 5 % 2
```

1.5 Conditions

Our language implements the regular `if` statement with optional (and possibly more than one) `elsif` and one optional `else`. `if` and `elsif` must be followed by a boolean which can be a variable, a logic operation, a comparison or a combination of those. Each of these blocks has it's own scope which is deleted when the code exits the `if` statement.

Logic operations Our language supports the three common logic operations: `not`, and and `or`, in this order of priorities, meaning that `not a or not b and c` is interpreted as `(not a) or ((not b) and (c))`.

Comparisons Comparisons support the following operators: ==, !=, >, <, <=, >=. The last four can only take integers on both sides of the comparison while the first two support any type on either side of the comparison. If they are the same type equality check will return True if the corresponding objects are equal (this is not a reference check) and inversely for inequality check. If they are not of the same type equality will always return False and inversely for inequality. Comparisons have priority over logic operations, meaning that `a < 3 and b != 2` is interpreted as `(a < 3) and (b != 2)`.

Examples

```
var = 0
if a:
    var = 1
elsif b < 12 and 4 != "Mordor":
    var = 2
elsif d and e or not (f == "Isengard"):
    var = 3
else:
    var = 4
end
```

1.6 Arrays

Arrays can contain a mix of all types as they are represented in memory as arrays of `Objects`. They are defined using brackets and commas (just like in Python). An empty array can also be defined using a colon (`[:n]`). The default value for the elements is then set to None.

Array accesses can be done with braces after the name of the variable. This can be used to access or reassign the value at a particular index in the array (`array[idx]`). If the index is larger than the length of the array, an `ArrayIndexOutOfBoundsException` will be thrown.

Examples

```
a = [:5]
a[3] = 2
a[2] = "They're taking the hobbits to Isengard"
b = ["You have", {"no":"power"}, "here"] # array containing a map object
a[5] = 2 # throws an ArrayIndexOutOfBoundsException
```

1.7 Maps

Our language also has a built-in data structure for dictionaries. It allows to have any primitive type (boolean, string or integer) as key and any type as value. This type of object can be initialized empty with just `{}` or by defining some keys and values with a colon separating the two.

To access elements of a map, one must simply use brackets and the key. If the user tries to access an element that is not present in the map, an error is thrown saying so. When adding new (key, value) pairs to a map, this is done via the equal sign: `map[key] = value`. If the key is already present in the map, the current value is overwritten, otherwise it is added and assigned to the value.

Examples

```
a = {2:True}
a[False] = None
a["Where was gondor"] = {1:2, 3:4} # map object as value of a map object
b = a["Viggo broke 2 toes when kicking that helmet"] # throws a RuntimeException
```

1.8 Loops

Our language supports while loops when the user needs to loop. It must have a condition (that has the same semantic as the `if` conditions) which is executed before running the block which is of course only run if the condition evaluates to `True`. The block creates a scope which is deleted when the code exits the `while` statement.

Examples

```
i = 0
b = 0
while i < 10:
    b = b + i*i
    i = i + 1
end
```

As an extension (also mentioned in the extensions section) we implemented a for loop structure to iterate over arrays. This initializes a variable at each execution of the array to be the next element in the array in the for statement. Just like for the `while` loop there is a scope for the code block but we also created an intermediary scope that only contains the iterative variable that we override at each iteration.

Examples

```
array = [1, None, "this"]
for a in array:
    println(b)
end
```

1.9 Function definition

Function definition has the following syntax. The keyword `def` must be followed by the function name, braces and its arguments (without typing). Then follows the code block and the end keyword. When a function is called the code block is executed in a new scope and the parameter variable names are mapped to the values passed in the function call. The user can return from a function using the `return` keyword followed by a value. If the function does not return a value, `None` is returned by default.

Examples

```
def f(a, b, c):
    return a+b*c
end
x = 1
b = False
def g():
    if b:
        return x + 5
    end
end
x = g()
z = 0
if x != None:
    z = f(1, 2, 3) + g()
else:
    z = 2
end
```

1.10 Printing to stdout

To print on the standard output, we provide two built-in functions: `print` and `println`. Both of them take any type as argument and return `None`. As it's name indicates, the `println` function adds a new line character at the end of the string when printing it on the standard output.

Examples

```
print({1:3, True:None})
println("If I take one more step")
println([2, None, False, "this"])
print(open("file.txt", "w"))
```

1.11 Program parameters

The user has access to program parameters in the form of a variable named `args` which is initialized at the start of the script as an array of strings containing program arguments.

Examples

```
print(args[0])
b = args[1]
```

1.12 Parsing

Parsing is done with the `int(s)` built in function. This function only accepts a string as an argument and returns the value of the integer. If the argument is not a string, it throws a `RuntimeException` and if the `parseLong()` java call fails, it throws a `NumberFormatException`.

Examples

```
a = int("Gollum")   # throws a NumberFormatException
b = int(3)          # throws a RuntimeException
c = int("1000")
```

2 Extra features

2.1 Expansions on minimal requirements

We chose to implement 2 types of loops : for loops and while loops. We also implemented the `elsif` statement, which allows for easier code writing. These added features are described in the section 1 of the report (Minimal requirements).

2.2 None

In our language we defined the special None value that has no real type (or is the only value of its type). It can be assigned to any variable, can be a value in a map (not a key however) and can appear in an array.

Examples

```
a = None
b = {1:None, True:False}
b["this"] = None
b[None] = 3 # throws a RuntimeException
c = [None, 2, 3]
```

2.3 Built-in functions

In addition to the functions mentioned in other places, we've built in 4 functions : `range(n)` returns a list of integers going from `0` to `n-1`, `len(a)` returns the length of the array/map given as argument, `indexer(a)` returns the keys to access the values stored in a (in the case of an array, it is similar to `range(len(a))`, in the case of a map it returns a list of the keys of said map) and `sort(a)` only accepts arrays as arguments, and returns a sorted copy of that array.

Examples

```
a = range(10)
b = len(a)
c = {"PO":2, "TA":2, "TOES":4}
for idx in indexer(c):
    print(c[idx])
end
d = [3, 2, 4, -1, 7, 2, 8]
d = sort(d)
```

2.4 List comprehension

As another additional feature we implemented list comprehension in the NS language. It allows to stream through arrays, creating a new array with an expression that can depend on an element of the initial array and an optional condition can be added to determine if the current element should be added (filtering). This required to create a scope for this expression and to initialize a variable for each element of the initial array.

Examples

```
a = [2, 3, 5, 7, 9]
b = [x % 3 for x in a if x > 6]
c = [x * 2 for x in [2+y for x in b]]

def f(x):
    if x:
        return 3
    end
    return 1
end
d = [f(x) for x in [True, False, None, True] if x != None]
```

2.5 Final variables

We have added final variables to our language in the following way. When assigning a value to a variable, the `final` keyword can be added in front. When done so, every time a user tries to assign a new value to a variable that was previously marked as final, a semantic error is thrown.

Examples

```
a = 3
final a = 5
a = 2 # Semantic Error
final b = [1, 2, 3]
b[2] = None # Semantic Error
```

2.6 I/O

The I/O uses 4 built-in functions and 1 new type. `open(filename, mode)` opens a file in read or write mode, `close(file)` closes that file, `read(file)` reads a line from the file (throws an `IOException` if the file is opened in write mode and returns `None` if the end of the file is reached) and `write(file, object)` writes the string representing the object to the file (throws an `IOException` if the file is opened in read mode).

Examples

```
fr = open("input.txt", "r")     # mode can be "r" or "read"  for reading
fw = open("output.txt", "w")    # mode can be "w" or "write" for writing

line = read(fr)
while line != None:
    write(fw, line)
    line = read(fr)
end

read(fw)                                # throws IOException
write(fr, "One ring to rule them all")  # throws IOException

close(fr)
close(fw)
```

3 Observations

Nullable and generic types Since our language has no explicit typing, generic types are irrelevant (all data objects are polymorph). They are also all nullable in the sense that `None` can be assigned to any variable, any array element or any map value (not key).

Negation of integers In arithmetic operations, negation has priority over addition and subtraction so `1 + -3` is a valid expression and is interpreted as `1 + (-3)`. It does not however have priority over multiplication, division and modulo operations so `5 % -4` or `5 / -2` are invalid and `-5 * 2` is interpreted as `- (5 * 2)`

Expressions & statements In our language, an expression can be a statement, meaning that `1 + 2` is a valid statement. This does not have any real effect on the program. However, if the last statement is an expression then this value is returned in the `interpret` function. This rendered our testing easier and could be useful if we wish to have a console to interpret our language and quickly see expression values.