LINGI2132 - Languages and Translators
Project 2 - Semantic Analysis

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

Group AC

Thursday 1st April, 2021

1 Introduction

The main feature of our language affecting the semantic analysis is the fact that it is dynamically typed and that it does simply not allow to specify a type for a certain variable. This is discussed in further detail in the following sections.

2 Expressions

Primary Types Integer, String, Boolean and None literals are immediately typed with the corresponding constant.

Arrays and Maps For arrays we have imposed that all elements should be of the same type. Therefore, when initializing an array with a literal, we check that all elements are of the same type (or that they are of unknown type or None type which is assignable to any type).

When accessing an element of an array, we check that the argument between brackets is indeed of type `INTEGER` (or `UNKNOWN_TYPE`).

Similarly, for maps all keys must be of a same type and all values must be of the same type within a same map. When accessing an element of a map, we only check that the argument is not of `NONE` type

Identifiers When we encounter an identifier node, we first lookup its declaration in the scope, if it is found we set its type, declaration and scope. If it is not we send an error to the user if either the identifier has not been declared before or if the identifier was declared after being used.

Functions In our language functions are defined without parameter types. Hence these identifiers will be assigned the `UNKNOWN_TYPE`. This means that some operations cannot be type checked and this will have to be done at run time.

We also made the choice to not impose a return type to a function when declaring it. Therefore a function call always has `UNKNOWN_TYPE` as type. However, when encountering a function call we do check that the call and function definition do have the same number of arguments.

For the reason stated above, there is no type checking for the expression in the return statement. The only check made is that the return statement is made within the scope of a function. We also made the choice to allow functions to not have a return statement. They will therefore default to returning the value `None`.

When defining a function, we create a scope to declare its parameters and since the FunctionNode contains a BlockNode, another child scope will be created there.

Built-in functions Our language has some built-in functions. The `range(n)` function only accepts integers as inputs (and returns an array containing all integers from `0` to `n-1`), `len(a)` accepts maps and arrays and returns an integer (the length of `a`).

`indexer(a)` and `sort(a)` both return arrays, but the former accepts maps and arrays and returns a list of the keys (`indexer(a)` will be functionally equivalent to `range(len(a))` if a is an array), while the latter only accepts arrays and sorts a copy of the array before returning it.

`int(s)` only accepts strings as input and will return the corresponding integer (it's type is thus an integer) while `print(a)` and `println(a)` accept all types of input and don't return anything. The return type for these two functions is thus `NONE`.

Operations Arithmetic operations (`+`, `-`, `*` ,`/`, `%` and negation) only accept integers as inputs and output another integer while logic operations are performed on booleans only, and the semantic analysis will thus not accept other types for `or`, and and `not`.

Comparisons Equality comparisons (== and !=) can be used with all types while inequality comparisons (<, >, <= and >=) can be used with integers and strings.

3 Statements

If When encountering an if block, the only check made is that the condition is of `BOOLEAN` type. Indeed, the if statement contains a BlockNode which will create a new scope and check that the statements and expressions contained in it are valid. Same applies for else statements (if they are `elsif` statements we also check the condition is of `BOOLEAN` type).

While Just like for the if block, when encountering a while block we check that the condition is of `BOOLEAN` type.

For A for loop in our language is similar to the ones in Python. This means that the for loops iterates over the elements of an array. The syntax is thus `for v in a : [block] end`, where v is a variable name (which will be bound to each value of an a) and a is of `ARRAY` type. When encountering a ForNode we create a new scope to add the variable that will contain elements of the iteration and another scope will be created for the BlockNode contained inside the ForNode.

4 Changes with Part 1

Inequality comparisons In the previous submission, we only allowed for inequality comparisons (<, >, <= and >=) between integers. Since we want to be able to compare strings too, we modified the parser to accept both strings and integers.

Relaxed parser We also made the parser generally less strict in different places. Indeed in the previous part of the project we tried to make our grammar relatively strict in the sense that we tried to, as much as possible, to avoid typing problems. However we realised that this rendered the grammar unnecessarily complex and that the errors generated were not very explicit. This was much better handled in the semantic analysis part as error messages can be much more explicit and helpful.

RootNode and other wrappers We also added the following nodes in the AST:

- RootNode: This addition allowed us to create the RootScope in which all reserved functions and primary values (`True`, `False`, `None`, `args`) are defined.
- BlockNode: This was added as a wrapper around lists of AST nodes as this custom class allowed us to register a function in the walker and create a scope for a given block of statements.
- ParameterNode: This is a wrapper around an IdentifierNode which allow us to treat parameters of a function as declarations of local variables as function parameters are declared with a simple variable name.