Symbol Table

xbudin05, xpojez00

October 15, 2021

1 Introduction

We are going to discuss our idea of implementing the symbol table for project (in Czech). We have decided to go for implementation using hash map rather than binary tree. Binary tree would solve problem of hash map, where multiple identifiers could possibly receive the same hash but for the cost of extensive access time. We are also facing problem with multiple scopes in the code, where the deepest scope variable is covering all above. To save identifiers we have a lower part of the structure, which is Hash table and below and the upper part to structure the scope we are working in and all identifiers declared in it. Picture for clearance beneath.

1.1 Conventional solution

Usually, it is implemented as a stack of hash maps, where each element in stack is new hash map, to solve the problem of scope covering. One element of hash map might then be a list to solve problem of overlapping hashes. This solution is pretty easy to understand but we have decided to improve this solution and save memory by creating always only one hash map and having a few more other structures. Also as a benefit we believe, we have also saved time in searching the deepest declaration of a certain variable, where it should be O(1)[1] no matter the stack dept. In the conventional solution this time corresponds to O(n), where n is the depth of stack.

2 Implementation of hash table

2.1 Our initial solution

We have decided to address both problems, multiple hashes of different identifiers colliding and having multiple declarations of variable, with one ordered double linked list. Idea was to create hash map, where every element is ordered double linked list, where the same identifiers are ordered according to their respective scopes. This idea can be implemented in a more clear way but the idea is preserved.

2.2 Our current solution

We have opted for creating list, where each element is holding the name of the identifier and pointer to the stack[2] for this specific identifier. This list is not ordered. Here is also clear why we have decided to implement symbol table with hash map, where accessing mostly only lists with size one is instant and does not consume much memory. This list solves our first problem with multiple identifiers having the same hash value. For the second problem of finding the deepest variable, we have chosen stack, as it is one of the best structures considering we only need push, pop and top for accessing variable in the top scope, which is covering all above.

2.3 Element

Approximate definition of our Symbol element, where this structure points to string, which is saved only in the hash map's list element. This way we can avoid duplicating name for later free-ing of this

[1] We do not consider overlapping hashes
[2] Explained later why this abstract structure

string. Saving it only once and pointing to this string also helps with time saved upon not recreating and free-ing this string. Type lets us decide, what type of Element we have and how we should read data. More variables might be added in the final implementation.

```
struct Element {
    Type type;
    char* name;
    bool isDefined;
    void* data;
}
```

3 Implementation of Symbol table

3.1 Solving Scopes

To implement scopes themselves, we have decided to choose a stack for it's simplicity. Whenever we need to clear scope we just clear all content of the top element. All identifiers in the scope are saved in list, where each Element is a pointer to the Element saved in the Hash table. This way, when clearing the list, we can navigate from the Hash table down to the Element itself, as there is deterministically[3] only one Element matching.

3.2 Buffer

We have also implemented a buffer, which saves every scope, that we might want to preserve, but we want it to be removed from the hash map and stack, holding current state of scopes. We remove[4] Element from the hash map, pop stack holding current state of scopes and save this poped list[5] into the buffer.

4 Summary

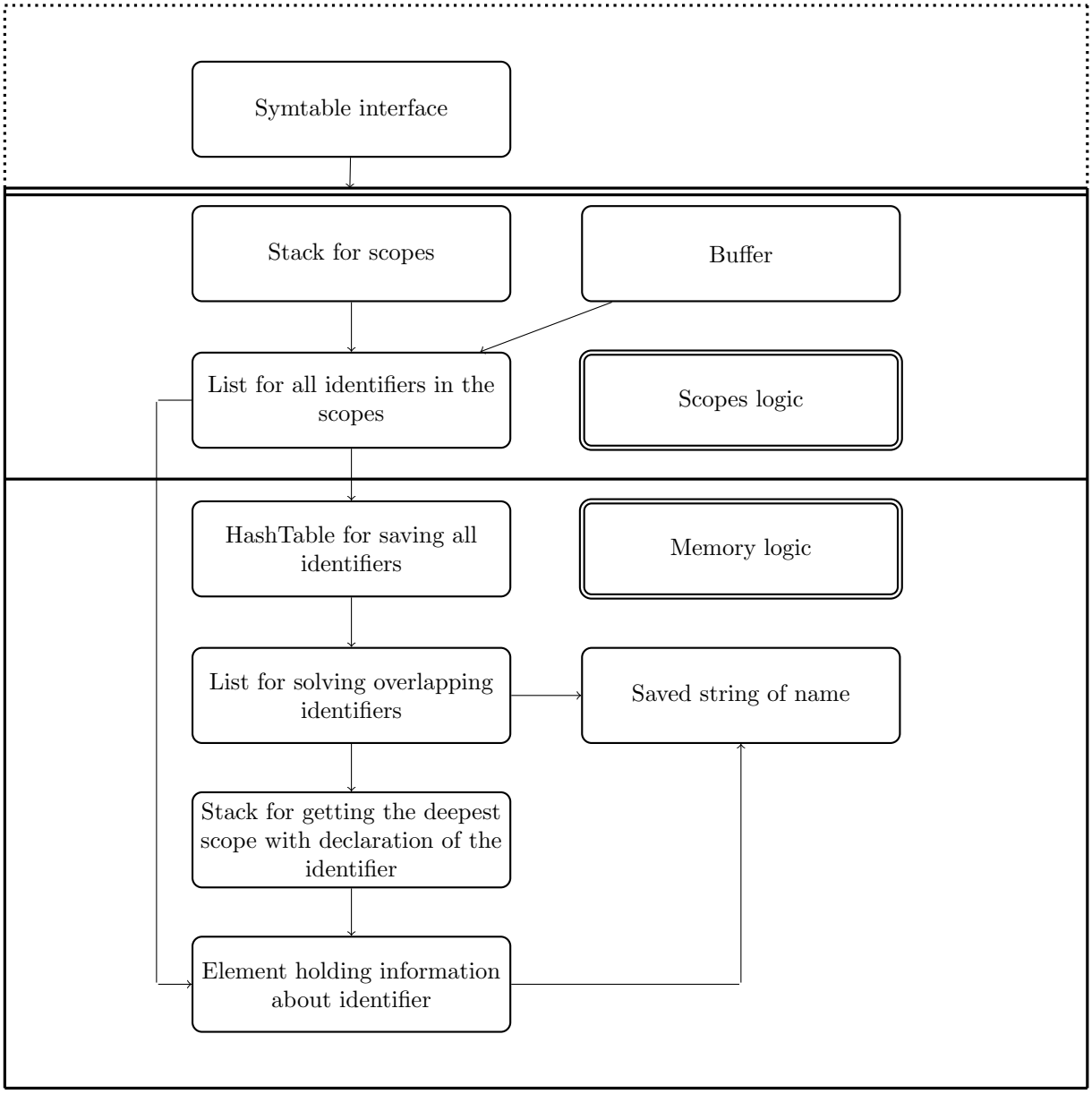
It is pretty apparent we have chosen a unique approach of harder implementation but for vast majority of cases instantaneous access and much better memory performance[6]. We have tried to explain idea as transparent and easy to understand as each problem is solved by separate structure layer.

[3] Every element can be destroyed only when clearing this list

[4] only the pointer

[5] This is the same list from stack, so Element can only be destroyed when clearing this list

[6] practical performance, in theory, if we would have number of identifiers equal to the size of hash map, with only single scope, we would use double the memory compared to the Conventional solution


Symtable interface
Stack for scopes
Buffer
List for all identifiers in the scopes
Scopes logic
HashTable for saving all identifiers
Memory logic
List for solving overlapping identifiers
Saved string of name
Stack for getting the deepest scope with declaration of the identifier
Element holding information about identifier