Symbol Table

xbudin05, xpojez00

October 11, 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 via 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.

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 very same solution 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[1] 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 need only push, pop and top for this new scope.

2.3 Element

Approximate definition of our Symbol element, where it can have its name pointing to the first list, which finds correct identifier and saves their name. This way we can avoid duplicating name. Type lets us decide, what type of Element we have and how we should read data.

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

[1] Explained later why this abstract structure

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 points to the Element saved in the Hash table. This way when clearing the list by poping the top of stack, we can navigate from the Hash table down to the Element itself, as there is deterministically only one Element matching.

4 Summary

It is pretty apparent we have chosen approach of higher memory demand but for vast majority of cases instantaneous access. We also believe, this implementation is transparent and easier to understand as each problem is solved by separate structure layer. This also allows for easier change of their implementation for future upgrades.