COSC 4785
Symbol table Lab Notes.

1 The Table

The symbol tables are some implementation of storage that allow rapid look-up of entries and are flexible in the number of entries that they can contain. In C++ you could reasonably use `std::unordered_map` class for the basis of the table object. Or just `std::map` if you want. I do NOT suggest that you write your own.

Regardless, you will have to implement functions that return the *type* of the identifier and/or whether or not the identifier is in the table or the table hierarchy. This is much easier with C++ as you can have public functions that provide a nice interface and private (protected) functions that internally manipulate the map.

Each table should have a *parent* pointer that can be used to access the next higher table in the scope structure. Do not forget that there is also a global table that contains the datatypes; int, void, **and all** the classes that are declared.

The global table does not need to be the parent of any other table. In fact, if you consider the situation, it might be wise to ensure that this global *type* table is NOT a parent of any other table. The global table **does** need to exist all the time the parser is running (never deleted) because as we declare new datatypes (classes), they have to be inserted into the global table. And those entries probably need some type of reference to the table created while that class is being processed.

Whether each table has pointers to its *children* or not is up to you. It may make some things easier when we do type checking and code generation. However, the table that is being used for the current code scope (class or method) must be available in addition to the global table, both for look-up of identifiers and for insertion of new variables (or methods). Speaking of methods, there is something you need to remember. The formal parameters of a method are also local variables of that method.

2 The Entry

The symbol tables are basically mappings from some identifier to a set of attributes that we normally call the *type*. The representation of this type can be done in multiple ways. For basic (or atomic) types like **int**, the name is the type. What *int* really means is part of the underlying grammar rules that cannot be expressed in any CFG. Classes are generally grouped under this definition even though they are a combination of language elements and possibly other types.

For complex types such as methods, the representation is somewhat less specific. For instance, a method with the signature

int foo(int a, x_class b, y_class c)

can be thought of as of type "$int \leftarrow int \times x\_class \times y\_class$".

All this means is that the entry in the symbol table must contain all the necessary information. You can figure out some way to encode the information. But, you have to make sure that your scheme is robust enough that it will not break. You might want to look at "C++ name mangling" on the Web for some ideas. There is another but here. Most of the examples will talk about things like *char*, *int*, and *void* datatypes, but how do encode the `x_class` datatype?

You might also want to consider whether the data for the entry is stored as a string (of some type) or as an object/struct. The string is easy but requires parsing, the object may be easier.

3 For the Future

I keep mentioning it but I do not know if you are taking note or not. In the future, you will have to completely type check the input and generate 3-address code. There is no reasonable way we will get to code optimization in this course.

In order to do this, you will have to completely validate the input (perform all the **semantic** checks). You will also need all the information about a class or method in order to instantiate the object and call the method. If you do not remember your COSC 2150, now would be a

good time to brush up on it a little and think ahead.

You should be asking yourself, "What information am I going to need in order to (eventually) convert the input to assembly language". And then make preparations for implementing those things in the next assignments. I strongly recommend that you do not add features that you are not ready to use, but you should lay the ground work, if possible.