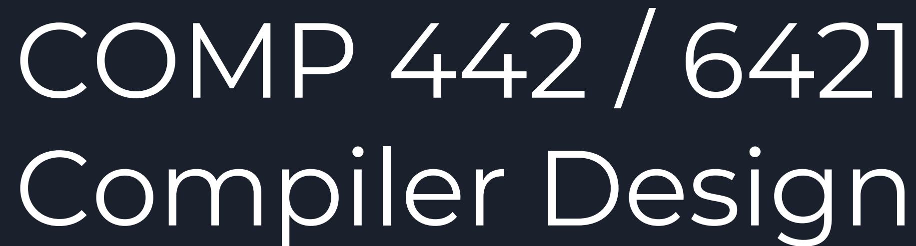


Tutorial 5

Instructor: Dr. Joey Paquet paquet@cse.concordia.ca

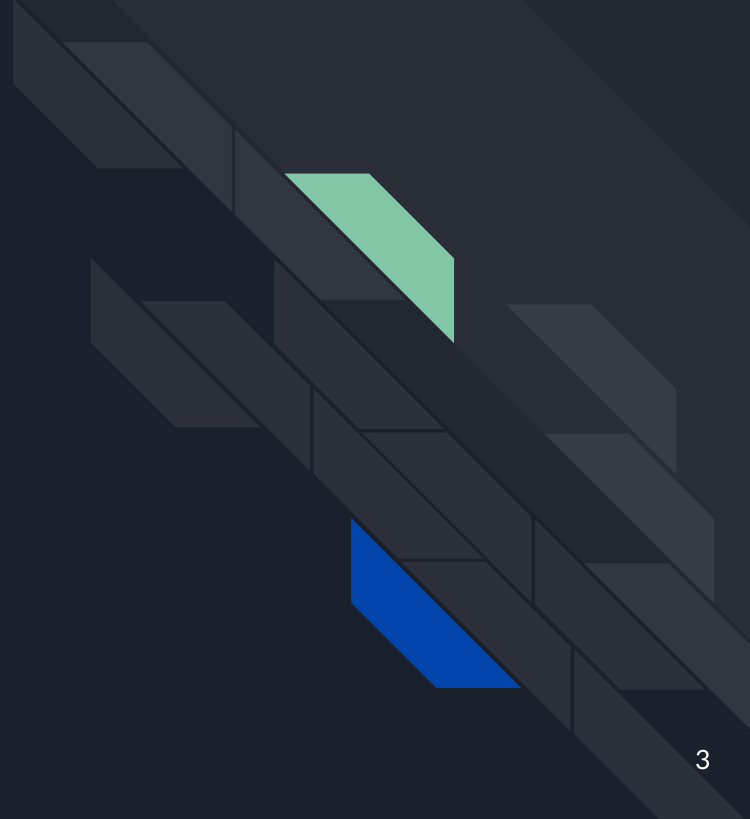
TAs: Haotao Lai h_lai@encs.concordia.ca

Jashanjot Singh s_jashan@cs.concordia.ca

Content

- Symbol table
- Visitor pattern
- Example of symbol table generation with visitor pattern apply to AST
- Circular dependencies check

Symbol Table



Symbol table is an important data structure used for scope manage, variable verification, type checking and code generation ……..etc

- Why we can’t assign an integer value to an string variable?
- Why we can’t access variable declared in different scope?
- Why we know how many memory we need to allocate for a declared variable?
- … …

Because we have symbol table! What kind of information we should store in a symbol table?

How a symbol table looks like?

```
1   class MyClass {
2       int x[3][8];
3       int addNum() {
4           int x;
5       };
6   };
7
8   program {
9       int x;
10      int y;
11      MyClass myClass[4][5];
12      MyClass myClass1;
13
14  };
15
16  int ff(int x, int y[2][2]) {
17      int xx;
18  };
```

Table Name: global table, Parent Table Name: null

name	kind	type	offset	link
MyClass	Class	MyClass	0	MyClass table
ff	Function	Int	0	ff table
program	Function	Null	0	program table

Top Level

Note: forget about the offset so far, will introduce in the next assignment

```
class MyClass {
    int x[3][8];
    int addNum() {
        int x;
    };
};

program {
    int x;
    int y;
    MyClass myClass[4][5];
    MyClass myClass1;

};

int ff(int x, int y[2][2]) {
    int xx;
};
```

Secondary Level

Table Name: MyClass table, Parent Table Name: global table

name	kind	type	offset	link
addNum	Function	Int	96	addNum table
x	Variable	Int[3][8]	0	null

Table Name: program table, Parent Table Name: global table

name	kind	type	offset	link
myClass1	Variable	MyClass	1928	MyClass
x	Variable	Int	0	null
myClass	Variable	MyClass[4][5]	8	null
y	Variable	Int	4	null

Table Name: ff table, Parent Table Name: global table

name	kind	type	offset	link
xx	Variable	Int	16	null
@returnAddr	Variable	Int	8	ff table
x	Parameter	Int	0	null
y	Parameter	Int[2][2]	4	null
@prevFp	Variable	Int	12	ff table

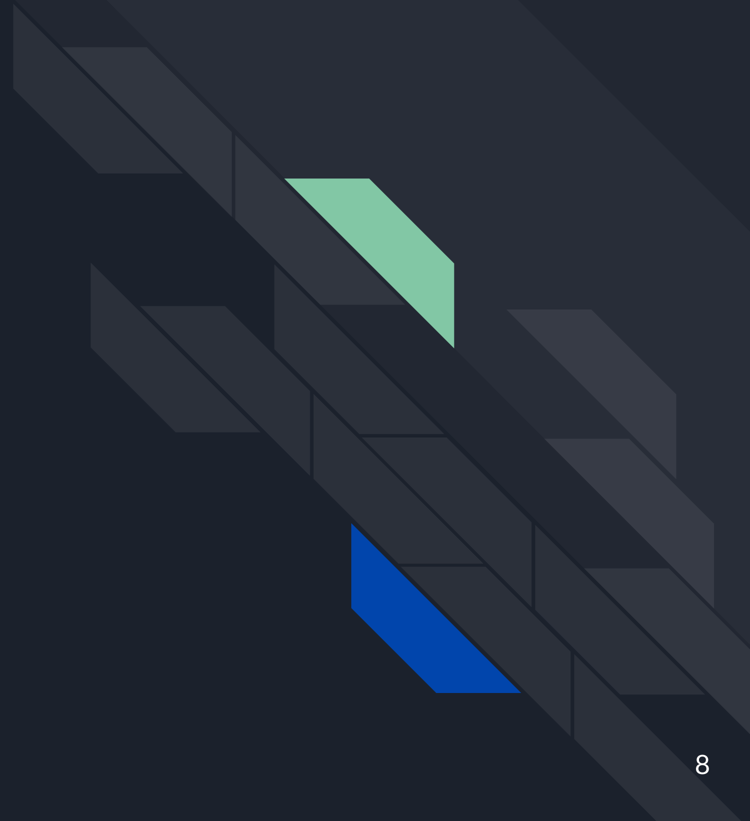
```
class MyClass {
    int x[3][8];
    int addNum() {
        int x;
    };
};

program {
    int x;
    int y;
    MyClass myClass[4][5];
    MyClass myClass1;

};

int ff(int x, int y[2][2]) {
    int xx;
};
```

Table Name: addNum table, Parent Table Name: MyClass table

name	kind	type	offset	link
@this	Parameter	MyClass	8	MyClass
@returnAddr	Variable	Int	0	addNum table
x	Variable	Int	12	null
@prevFp	Variable	Int	4	addNum table

Third Level

Visitor Pattern



Design Pattern

Design pattern is a general reusable solution to a commonly occurring problem when we design a software.

One fact is that without design pattern, we can still write code and it may work properly but with design pattern we can write more reusable, maintainable, robustic code.

Design pattern is something existing in the world not being invented by anyone but it becomes popular after “GoF ” publish their book which conclude total 23 patterns.

Visitor pattern is one of these 23 cataloged into behavioral pattern.

Visitor Pattern

It should be possible to define a new operation for (some) classes of an object structure without changing the classes.

When new operations are needed **frequently** (visit node in the AST) and the object structure consists of many unrelated classes, it's **inflexible** to add new subclasses each time a new operation is required.

Sum up the demand up (for the project)

1. Want to visit the AST nodes for different purpose (execute different operation on the same node)
2. Don’t want to change the structure of the AST Node (hard to maintain, easy to mess up)

Visitor Pattern

```
1  Visitor
2      + visit( ElementType )
3
4  Element
5      + accept( VisitorType )
```

```
35  JoeyHouse : Element
36      + accept( FriendVisitor )
37      + accept( StudentVisitor )
38      ... ...
39
40  EricHouse : Element
41      + accept( FriendVisitor )
42      + accept( ClassmateVisitor )
43      ... ...
44
45  JashHouse : Element
46      + accept( FriendVisitor )
47      + accept( ClassmateVisitor )
48      + accept( RelativeVisitor )
49      ... ...
50
```

```
14  FriendVisitor : Visitor
15      + visit( JoeyHouse )
16      + visit( EricHouse )
17      + visit( JashHouse )
18
19  StudentVisitor : Visitor
20      + visit( JoeyHouse )
21      + visit( EricHouse )
22      + visit( JashHouse )
23
24  ClassmateVisitor : Visitor
25      + visit( JoeyHouse )
26      + visit( EricHouse )
27      + visit( JashHouse )
28
29  RelativeVisitor : Visitor
30      + visit( JoeyHouse )
31      + visit( EricHouse )
32      + visit( JashHouse )
33
```

Visitor Pattern

```
1   Visitor
2       + visit( ElementType )
3
4   Element
5       + accept( VisitorType )
```

```
63  EricHouse : Element
64      accept( Visitor friend )
65      {
66          friend.visit(this)
67      }
68
69      accept( ClassmateVisitor classmate)
70      {
71          classmate.visit(this)
72      }
```

```
46  FriendVisitor : Visitor
47      visit(JoeyHouse joeyHouse)
48      {
49          // greeting using "bonjour"
50      }
51      visit(EricHouse ericHouse)
52      {
53          // greeting using "nihao"
54      }
55      visit(JashHouse jashHouse)
56      {
57          // greeting using "hello"
58      }
```

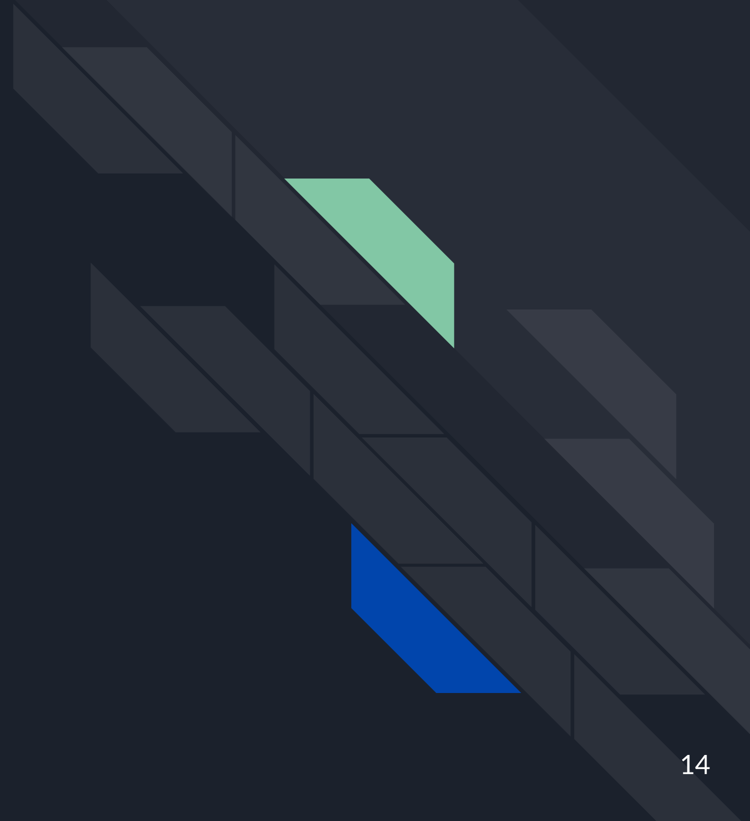
Theoretical Level

- Visitor
  - ConcreteVisitor
- Element
  - ConreteElement

Map to our project

- Visitor
  - SymbolTableGeneratoinVisitor
  - TypeCheckingVisitor
  - CodeGenerationVisitor
- AstNode
  - ProgNode
  - ClassNode
  - …………

Example



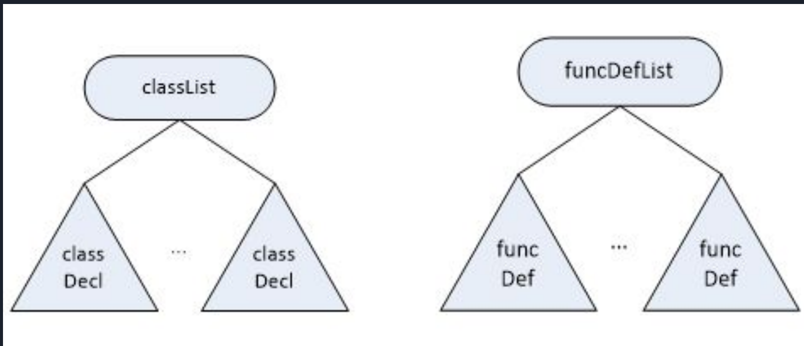
Useful facts

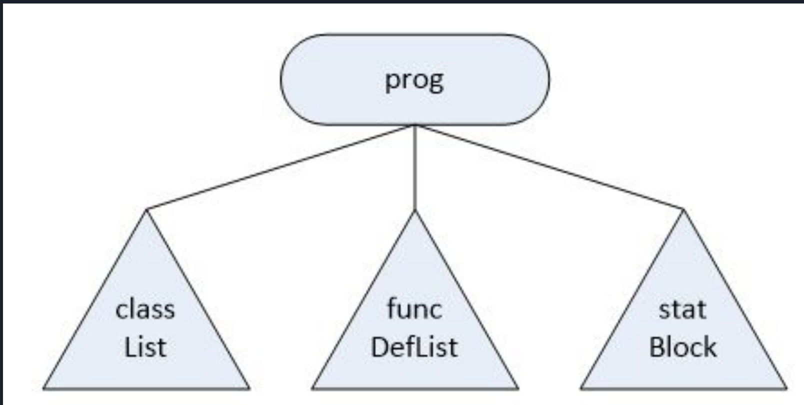
According to our grammar given in assignment 2:

1. All classes must be declared before the main function ("program" function)
2. All functions (both free functions and member functions) must be defined before the main function and after the classes declaration;

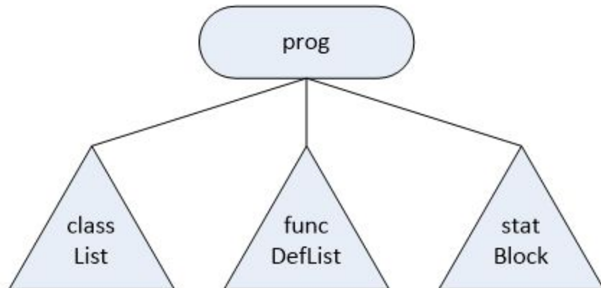
[first-pass] In-order traverse the AST (assume you already have an AST)

1. Create classes' symbol table
2. Create functions' symbol table
   a. Free function we don't do any link so far
   b. Member function should be linked with its class
3. Create a global table for "program" function
   a. Add all classes to the global table
   b. Add all free function to the global table
   c. Add the program function itself to the global table

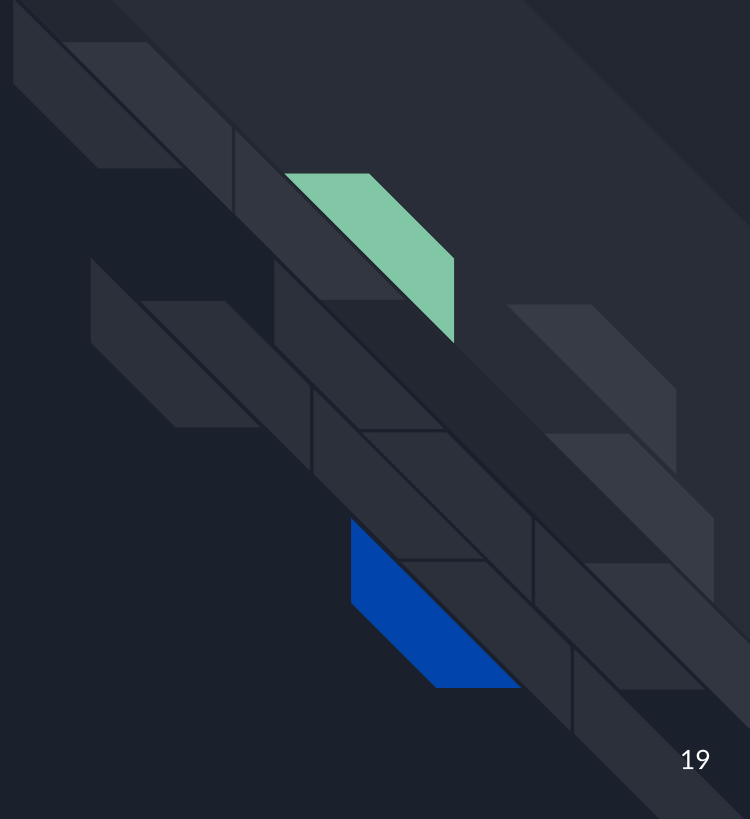

classList
class
Decl
...
class
Decl
funcDefList
func
Def
...
func
Def


prog
class
List
func
DefList
stat
Block

```java
public class SymTabCreationVisitor extends Visitor {
…
  public void visit(ProgNode node){
    node.symtab = new SymTab("global");
    // for classes, loop over all class declaration nodes
    for (Node classelt : node.getChildren().get(0).getChildren())
      //add the symbol table entry of each class in the global symbol table
      node.symtab.addEntry(classelt.symtabentry);
    // for function definitions, loop over all function definition nodes
    for (Node fndefelt : node.getChildren().get(1).getChildren())
      //add the symbol table entry of each function definition in the global symbol table
      node.symtab.addEntry(fndefelt.symtabentry);
    // for the program function, get its local symbol table from node 2 and create
    // an entry for it in the global symbol table
    // first, get the table and change its name
    SymTab table = node.getChildren().get(2).symtab;
    table.m_name = "program";
    node.symtab.addEntry("function:program", table);
  };
}
```



Circular Dependencies



Assume you have two class: A and B

```
class A                class B
{                      {
        B b;                   A a;
}                      }
```

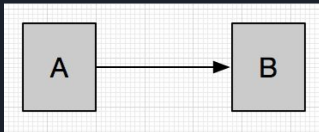
How can you know the size of A or the size of B?
(size → how many memory you need for each object of type A or type B)

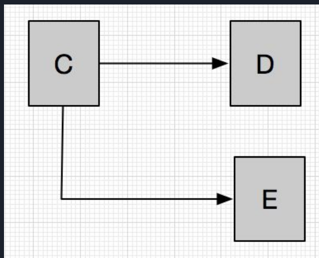
In real world programming language like C/C++, the solution for that problem is we are using pointer instead of object.

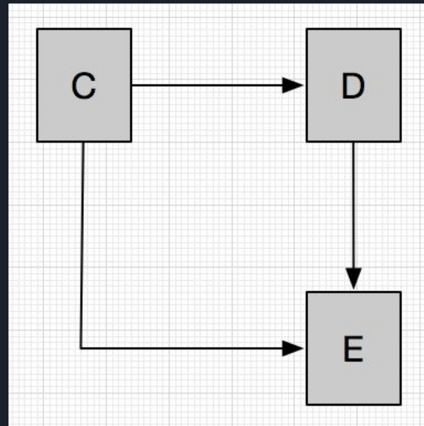
Example (1)

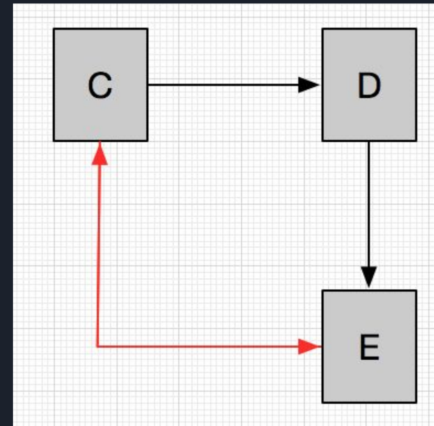
Assume in your source file, you have the following classes (arrow means dependency):

```
class A : B          class C : D, E        class D : E          class E : C
{                    {                     {                    {
        B b                  … …                   … …                  … ...
}                    }                     }                    }
```









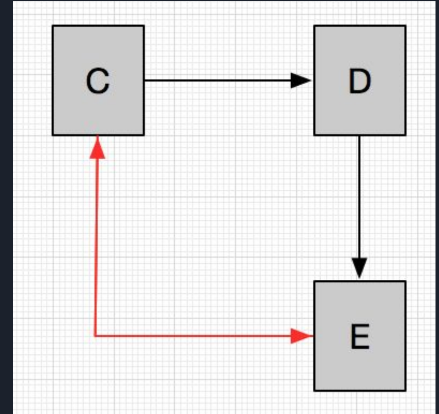
Example (2)

```
class A : B
{
        B b
}
```

```
class C : D, E
{
        … …
}
```

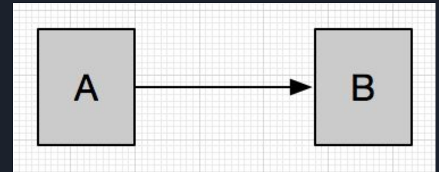
```
class D : E
{
        … …
}
```

```
class E : C
{
        … ...
}
```



If you have a circle in your graph, it means you have circular dependencies.

Circular dependencies is not allowed in our project, since the language we defined doesn't have pointer.



Just from what we know now, it is obviously can be implemented as following:

1. Build the directed graph
2. Search the graph to see whether there is a circle

If you think the graph implementation is so complicated, we can have a list implementation

1. Build dependencies list for each class
2. Substitute the class with its dependencies you get later
3. Keep substitute until you can’t do it anymore
4. Check if there is repetition of any class

Example of list implementation

Thanks!

- You are not forced to use visitor pattern, but it is recommended and you can get support from Prof. and TAs;
- The circular dependencies check will only be assigned a tiny point you should have your own plan to determine implement which features;