* To start using ‘Extractor’, first we need to create an extractor object using its constructor.
* After that source code can be added to that extractor object using addSource(path) method. Path is relative to the extractor’s root directory.
* ‘addSource’ function returns a source code object containing various data.
* ‘root’ attribute in source object is the entry to the source code tree saved in source code object.
* Following code lines shows how we can access root of the source code tree.

extractor = Extractor()  
sourceObj = extractor.addSource(**"omp\_hello.c"**)

root = sourceObj.root

**Structure of the source code tree**

* Source code tree is basically a set of nodes which are of type ‘Block’. (directly or inherited)
* Tree is organized such that, no portion of the source code is stored in multiple nodes.
* Since for loops and directives are the main focus of optimization, for loops and directives are stored in a special containers named as ‘ForLoop’ and ‘Directive’ respectively.
* Directive objects are always stored in a special container called ‘StructuredBlock’. Following is an example structured block.

#pragma omp parallel for schedule (static,8)

PARAMETERS

CLAUSE

DIRECTIVE NAME

STRUCTURED

BLOCK

DIRECTIVE

HEADER

for (int i = 1; i < 100; i++)

{  
 //loop body

LOOP

BODY

//can contain nested Structured Blocks

}

* Directives under consideration can be of three types.
  + directive followed by an associated for loop – similar as above example
  + directive followed by block enclosed in curly brackets such as ‘omp parallel’ directive
  + just the directive such as ‘omp target enter data’
* For each of these types body of the structured block can be a ForLoop object, Block object, or null in the above order.
* Further, for loops without an associated directive is also stored in a ‘StructuredBlock’ so that while optimizing, directive can be added accordingly. Especially helpful in nested threading scenarios to change the level of the directive.
* Block class has three main properties
  + body - string
  + elements – list
  + parent –‘Block’ object
* Every class inherited from ‘Block’ class uses these three properties to store various data according to their needs.
* Using above given structured block, value of each of these properties are in the following table. (assuming this is the only thing available in the source code)

|  |  |  |  |
| --- | --- | --- | --- |
|  | body | elements | parent |
| Structured Block | “” | [Directive, ForLoop object] | root |
| Directive | “#pragma omp parallel for” | [list of ‘Clause’ objects] | Structured Block |
| Clause | “schedule” | [list of ‘Parameter’ objects] | Directive |
| Parameter | “static” | [] | Clause |
|  | “8” | [] | Clause |
| ForLoop | “for (int i = 1; i < 100; i++)” | [‘Block’ object containing curly bracket area] | StructuredBlock |

* Every other insignificant portions of the code is saved in pure Block objects. For example include statements at the start of the code. Their parent is root node.

**Helpful Functions**

* These functions can be called on any ‘Block’ type object. As root is the entry point, we can start using them on root node.
* Following are three helpful functions implemented so far to get data from source code tree,
  + getParent() – returns parent Block of any Block
  + getNext() – returns next Block object in the order of the source code
  + getContent() - returns a string containing all the ‘body’ attributes of child Blocks. If called on root node, initial source code is returned
  + isChild() – checks whether a block is a child block of another block. Since root node’s parent is NULL someBlock.isChild(None) will always be true
* More helpful functions are to be developed so that optimizations can be implemented efficiently and easily.