Project Report Iteration 1

1. Development Plan

The below table illustrates the division of component development among team members. The asterisk (*) indicates that the team member contributed to development in that component. The components will be explained in further detail in Section 3: SPA Design.

	Iteration 1					
Component	Yung Chung	Jia Feng	Jun Kiat	Timothy	Cher Lin	Pheng
Parser				*		
Tokenizer			*			
PKB			*		*	*
FollowsTable						*
ParentTable						*
ModUsesTable					*	
ConstantTable			*			
StmtType Table					*	
Query Preprocessor		*				
Query Evaluator	*					

Table 1: Division of workload for Iteration 1

2. Scope of Prototype Implementation

We have implemented the iteration 1 prototype according to the requirements specified.

3. SPA Design

3.1 Overview

Our SPA prototype consists of 3 main components - **Front-End, PKB, and Query Processor.**

Front-End consists of the Parser which reads the user-specified SIMPLE source program and simultaneously populates the **PKB**. The **Query Processor** accepts user-queries, evaluates the query by extracting program design abstractions from the **PKB,** and subsequently projects the result to the user.

3.2 Design of SPA Components

a. SPA Front-End Design

Front-End consists of the **`Parser`** and **`Tokenizer`** classes.

Upon receiving a user-specified SIMPLE source file, the **`Parser`** creates an instance of the **`Tokenizer`** class. The **`Tokenizer`** breaks the SIMPLE file into string tokens of variable names, constants, and SIMPLE language flags such as =, *{*, and *while*. The **`Parser`** may then call **`Tokenizer::getNextToken()`** to receive the next string token in the sequential order it was written in the program.

The **`Parser`** operates by top-down recursive descent. The below code snippet illustrates the process:

```
void Parser::parseStmtLst()
{
        if (next_token == WHILE_FLAG)
        {
                parseWhileStmt();
        }
        else
        {
                parseAssignStmt();
        }
        match(SEMICOLON_FLAG);
        stmtLine++;
        if (next_token == RIGHT_BRACES)
        {
                return;
        }
        else
        {
                parseStmtLst();
        }
}
```

Observe that until right braces ("}") are seen, **`parseStmtLst`** is recursively called on each SIMPLE program line, and each **`parseStmtLst`** call itself branches out according to the statement type (*while* or *assign*) of that line.
As the **`Parser`** runs through the program, it maintains global variables: **`stmtLine`** and **`followsMaxNestingLevel`** to keep track of *stmt* count and the *stmtLst* count (which is used to identify Follows relationships) respectively. Additionally, there are two stacks: **`parentStack`** and **`followsStack`** to keep track of the parents (indirect and direct) as well as nesting levels respectively. The **`Parser`** simultaneously populates the **PKB** with design abstractions.

Upon entering each new program line, **`stmtLine`** is incremented. The statement's current nesting level and direct parent, obtained by peeking at the **`followsStack`** and **`parentStack`** respectively, is updated in the **PKB** (a NO_PARENT_FLAG of value -1 is used to indicate when a statement is not within any container).

If the current statement, which we will denote by S, is of *while* type, the following occur:

1. **PKB** is updated with the <S, *while*>statement type pairing
2. S is pushed onto **`parentStack`**, thus setting S as the current parent
3. **`followsMaxNestingLevel`** is incremented and pushed onto **`followsStack`**, indicating that we are entering into a new *stmtLst* (nesting level)
4. Control variable is used to update 'Uses' relationships in the **PKB**
5. **`parseStmtLst`** is recursively called to anticipate either a *while* or *assign* statement next
6. Upon the recursive call's return, **`followsStack`** and **`parentStack`** are popped to signal that we are exiting a container statement

If the current statement, S, is of *assign* type, the following occur:

1. **PKB** is updated with the <S, *assign*> statement type pairing
2. Left-hand side is used to update 'Modifies' relationships in **PKB**
3. Right-hand side is used to update 'Uses' relationships in **PKB**

The **`Parser`** eventually terminates when the original function call (**`parseProgram`**), having successfully reached the end of the outermost *stmtLst*, returns.

b. PKB Design

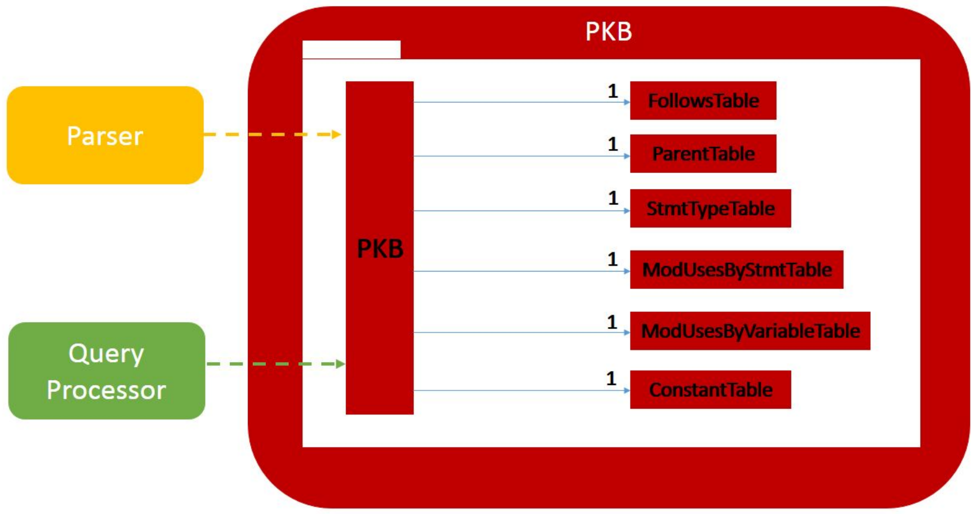

Figure 1: Structure of the PKB component

PKB stores the concrete information It consists of the **`PKB`** class interface, as well as **`FollowsTable`**, **`ParentTable`**, **`StmtTypeTable`**, **`ModUsesbyStmtTable`**, **`ModUsesByVariableTable`** and **`ConstantTable`** classes.

The **PKB** implements a Facade design pattern. The **`PKB`** interface receives the SIMPLE source design abstractions from the **Parser** and populates its tables appropriately. Likewise, it receives all requests from the **Query Processor** on behalf of the tables, makes a request to the relevant tables internally, then sends the results back.

The **PKB** and its various tables are never instantiated, instead, a public static pointer is created whenever **`PKB`** is first called which is obtained by all components with the **`PKB::getPKB()`** method.

The **`FollowsTable`** consists of <*stmtLine*, *nestingLevel*> key-value pairs stored in an STL list. Please pay attention to the *nestivgLevel*. For example, for any two statements, Follows*(s1, s2) holds if the below two conditions hold:

1. s1 and s2 are in the same *nestingLevel*
2. s1 and s2 are in sequential order (i.e. s1 < s2)

Follows(s1, s2) holds if the preceding two conditions are true, and that additionally, s2 comes immediately after s1 in that same *nestingLevel*.

The **`ParentTable`** consists of <*stmtLine*, *directParent*> key-value pairs. From this pairing, 'Parent' relationships can be immediately deduced, and 'Parent*' relationships can be determined by following the chain of parents up until the NO_PARENT_FLAG indicator is reached.

`StmtTypeTable` simply associates *stmtLine* to a statement type (*while* or *assign*).

ModUsesbyStmtTable and **ModUsesByVariableTable** store the 'Modifies' and 'Uses' relationships among variables and *stmtLines*. As the name suggests, the former is keyed by *stmtLine*, whereas the latter is keyed by variable names. The information stored in both tables are not mutually exclusive, and we will explain this design choice in Section 6: Discussion.

ConstantTable stores <*constant*, *list<int> stmtLine*> pairings, where Uses(s, constant) holds for each s in the *stmtLine* list.

c. Query Processor Design

The main functionalities of **PQL** is to parse the incoming queries and verify its syntax validity by ensuring that the syntax is correct with respect to the SPA requirement. A **PQLController** class, acting as a main controller, controls the sequences of actions of the whole **PQL** components. **PQLParser** acts as a filter to decompose the raw query into smaller entities. The smaller entities are further processed by **PQLVerifyTable** to check if the queries were valid.

PQLController:

PQLController is the controller class of the whole **PQL** system. It invoked **PQLParser** to further evaluate queries.

PQLParser:

PQLParser acts as a façade class for the whole PQL system. Its role is described as below:

1. Input query would be evaluated by splitting them into smaller string.
2. Each string would be preprocessed by removing extra spaces and conduct check to see if query syntax is aligned to the SPA grammar rule.
3. Queries would then pass to **PQLVerifyTable** to conduct more intensive checks to ensure its validity.

Below is an example how **PQLParser** would process query:

1) assign a; while w;
2) Select s such that Follows(1,s)

PQLParser would first read in first line, and it would split the string into two strings by looking for the semicolon. This newly initialized vector consists of two strings, which are "assign a" and "while w". These two strings would be further split by spaces. This would result in a vector of four strings – "assign", "a", "while", and "w". Lastly, the vector of four strings will be classified into the correct type. The second line of query would be processed in the similar fashion, except that after splitting the string into multiple subsets of strings, a function would be invoked to merge "(1," and "s)". The end result would look like:

Select	S	such	that	Follows	(1,s)

PQLVerifyTable:
As the name suggest, it is used to check the validity of the query. After pre-processed is done at the `PQLParser`, `PQLParser` would pass the string to `PQLVerifyTable` to conduct a more extensive checks, and classify the type of entities accordingly.
For instance, taking the following input string vector for `PQLVerifyTable` as an example:

Select	S	such	that	Follows	(1,s)

`PQLVerifyTable` would check whether the first string is Select. If it is not "Select", it will auto terminate, and no other check would be conducted. If the first string is "Select", the next string "S" would be evaluated to check if "S" is in the entity declaration. If "S" is not in the entity declaration, it will return false to `PQLParser` to indicate that the query is wrong. Else, "such" and "that" clause will be evaluated. Similar to earlier checks, if the query does not contain "such" and "that", a false result will be returned. If the check for "such" and "that" holds, the "Follows" clause would be check with the list of clauses in Iteration 1. Finally, "(1,s)" would be evaluated by first removing the front and end brackets. Depending on the type of clauses, it would either invoked `entRefCheck` or `stmtRefCheck`. For this case, its "Follow" clause, `stmtRefCheck` would be conducted to ensure both the inner content are stmt reference.

4. Documentation and Coding Standards

We use the *google-cppguide* as reference for our coding standards[1]. Below are some examples of the standards we adhere to:

a. **Naming Conventions :**
    i. Use meaningful names, avoid abbreviations.
b. **Comments :**
    i. Use // for single comments, and /* */ for a block of comments.
    ii. Comments should be provided for important or complicated functions.
    iii. Do not state the obvious. In particular, avoid literally describing what the code does, but instead, provide higher level comments explaining why the code does what it does.
c. **Presentations :**
    i. One statement per line.
    ii. Conditionals: always enclose body of conditional statements with curly brackets to avoid potential errors in the code.
    iii. Conditionals: if and else keyword should belong on separate lines.
d. **Formatting :**
    i. Each line of code should be at most 80 characters long.
    ii. Default indentation is one tab.
    iii. One space is added before and after each operator.
e. **Files :**
    i. All .cpp files have associated.h header files.

[1] https://google.github.io/styleguide/cppguide.html

5. Testing

```
TEST_METHOD(TestTokenizer1) {
	Tokenizer tk("SIMPLE_test_1.txt");
	list<string> expectedList;
	vector<string> expectedVec;
	expectedVec = { "procedure", "ABC", "{",
		"x", "=", "1", ";",
		"b", "=", "2", ";",
		"y", "=", "x", "+", "i", ";",
		"while", "i", "{",
		"while", "z", "{",
		"good", "=", "b2y", ";", "}",
		"apple", "=", "orange", ";", "}", "}" };
	vecToListHelper(expectedVec, expectedList);

	list<string>::iterator it = expectedList.begin();
	for (; it != expectedList.end(); ++it) {
		string token = tk.getNextToken();
		Assert::AreEqual(*it, token);
	}
}
```

Test purpose: To test if the Tokenizer class is able to tokenize the SIMPLE file correctly
Required Test Inputs: SIMPLE file
Expected Test Results: Contents of the SIMPLE file after being split into individual tokens

```
TEST_METHOD(TestParentTable) {
	Parser p("SIMPLE_test_2.txt");
	p.process();

	const int NO_PARENT = -1;
	const int FALSE = -1;

	// Parent(_, x), where x is not in any container
	// Should return NO_PARENT
	Assert::AreEqual(PKB::getPKB()->getParentOf(1), NO_PARENT);
	Assert::AreEqual(PKB::getPKB()->getParentOf(2), NO_PARENT);
	Assert::AreEqual(PKB::getPKB()->getParentOf(3), NO_PARENT);
	Assert::AreEqual(PKB::getPKB()->getParentOf(6), NO_PARENT);

	// Correct Parent relationships
	Assert::AreEqual(PKB::getPKB()->getParentOf(4), 3);
	Assert::AreEqual(PKB::getPKB()->getParentOf(5), 3);
	Assert::AreEqual(PKB::getPKB()->getParentOf(7), 6);
	Assert::AreEqual(PKB::getPKB()->getParentOf(9), 8);
	Assert::AreEqual(PKB::getPKB()->getParentOf(12), 11);

	// Incorrect Parent relationships (container stmt does not match parent)
	Assert::AreNotEqual(PKB::getPKB()->getParentOf(4), 6);
	Assert::AreNotEqual(PKB::getPKB()->getParentOf(7), 3);
	Assert::AreNotEqual(PKB::getPKB()->getParentOf(12), 8);

	// Incorrect Parent relationships
	// (not direct parent, i.e. Parent* holds but not Parent)
	Assert::AreNotEqual(PKB::getPKB()->getParentOf(10), 6);
	Assert::AreNotEqual(PKB::getPKB()->getParentOf(12), 6);
}
```

Test purpose: To test if the getParentOf(*stmtNum*) method works correctly
Required Test Inputs: SIMPLE file
Expected Test Results: The test case will check if the parent statement number returned by the method is correct and assert false otherwise

```
TEST_METHOD(testTwoClauseValidQuery) {
       string queryString = "assign a1b2c3; variable d4; while e5; constant c6; stmt
       salm0n3lla; Select d4 such that Follows*(a1b2c3, e5) pattern a1b2c3(\"y\",_\"10\"_)";
       QueryValidator qv(queryString);
       QueryTable qt = qv.parse();
       Assert::AreEqual(qt.isNullQuery(), false);
       Clause selectClause = qt.getSelectClause();
       Assert::AreEqual(selectClause.isClauseNull(), false);
       Clause suchThatClause = qt.getSuchThatClause();
       Assert::AreEqual(suchThatClause.isClauseNull(), false);
       Clause patternClause = qt.getPatternClause();
       Assert::AreEqual(patternClause.isClauseNull(), false);
       Assert::AreEqual(selectClause.getArg().at(0), (string)"d4");
       Assert::AreEqual(selectClause.getArgType().at(0), (string)"variable");
       Assert::AreEqual(suchThatClause.getRelation(), (string)"follows*");
       Assert::AreEqual(suchThatClause.getArg().at(0), (string)"a1b2c3");
       Assert::AreEqual(suchThatClause.getArgType().at(0), (string)"assign");
       Assert::AreEqual(suchThatClause.getArg().at(1), (string)"e5");
       Assert::AreEqual(suchThatClause.getArgType().at(1), (string)"while");
       Assert::AreEqual(patternClause.getRelation(), (string)"patternAssign");
       Assert::AreEqual(patternClause.getArg().at(0), (string)"y");
       Assert::AreEqual(patternClause.getArg().at(1), (string)"10");
       Assert::AreEqual(patternClause.getArg().at(2), (string)"a1b2c3");
       Assert::AreEqual(patternClause.getArgType().at(0), (string)"string");
       Assert::AreEqual(patternClause.getArgType().at(1), (string)"string");
       Assert::AreEqual(patternClause.getArgType().at(2), (string)"assign");

       queryString = "assign a1b2c3; variable d4; while e5; constant c6; stmt salm0n3lla;
       Select d4 such that Parent*(e5, salm0n3lla) pattern a1b2c3(d4,_\"on3l0ngstr1ng\"_)";
       qv = QueryValidator(queryString);
       qt = qv.parse();
       Assert::AreEqual(qt.isNullQuery(), false);
       selectClause = qt.getSelectClause();
       Assert::AreEqual(selectClause.isClauseNull(), false);
       suchThatClause = qt.getSuchThatClause();
       Assert::AreEqual(suchThatClause.isClauseNull(), false);
       patternClause = qt.getPatternClause();
       Assert::AreEqual(patternClause.isClauseNull(), false);
       Assert::AreEqual(selectClause.getArg().at(0), (string)"d4");
       Assert::AreEqual(selectClause.getArgType().at(0), (string)"variable");
       Assert::AreEqual(suchThatClause.getRelation(), (string)"parent*");
       Assert::AreEqual(suchThatClause.getArg().at(0), (string)"e5");
       Assert::AreEqual(suchThatClause.getArgType().at(0), (string)"while");
       Assert::AreEqual(suchThatClause.getArg().at(1), (string)"salm0n3lla");
       Assert::AreEqual(suchThatClause.getArgType().at(1), (string)"stmt");
       Assert::AreEqual(patternClause.getRelation(), (string)"patternAssign");
       Assert::AreEqual(patternClause.getArg().at(0), (string)"d4");
       Assert::AreEqual(patternClause.getArg().at(1), (string)"on3l0ngstr1ng");
       Assert::AreEqual(patternClause.getArg().at(2), (string)"a1b2c3");
       Assert::AreEqual(patternClause.getArgType().at(0), (string)"variable");
       Assert::AreEqual(patternClause.getArgType().at(1), (string)"string");
       Assert::AreEqual(patternClause.getArgType().at(2), (string)"assign");

       queryString = "assign a1b2c3, aquaman, as0n; variable d4,vict0ry; while e5; constant
```

```
        c6; stmt salm0n3lla; Select c6 such that Modifies(salm0n3lla, _) pattern
        as0n(vict0ry,_\"on3l0ngstr1ng\"_)";
        qv = QueryValidator(queryString);
        qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), false);
        selectClause = qt.getSelectClause();
        Assert::AreEqual(selectClause.isClauseNull(), false);
        suchThatClause = qt.getSuchThatClause();
        Assert::AreEqual(suchThatClause.isClauseNull(), false);
        patternClause = qt.getPatternClause();
        Assert::AreEqual(patternClause.isClauseNull(), false);
        Assert::AreEqual(selectClause.getArg().at(0), (string)"c6");
        Assert::AreEqual(selectClause.getArgType().at(0), (string)"constant");
        Assert::AreEqual(suchThatClause.getRelation(), (string)"modifies");
        Assert::AreEqual(suchThatClause.getArg().at(0), (string)"salm0n3lla");
        Assert::AreEqual(suchThatClause.getArgType().at(0), (string)"stmt");
        Assert::AreEqual(suchThatClause.getArg().at(1), (string)"_");
        Assert::AreEqual(suchThatClause.getArgType().at(1), (string)"any");
        Assert::AreEqual(patternClause.getRelation(), (string)"patternAssign");
        Assert::AreEqual(patternClause.getArg().at(0), (string)"vict0ry");
        Assert::AreEqual(patternClause.getArg().at(1), (string)"on3l0ngstr1ng");
        Assert::AreEqual(patternClause.getArg().at(2), (string)"as0n");
        Assert::AreEqual(patternClause.getArgType().at(0), (string)"variable");
        Assert::AreEqual(patternClause.getArgType().at(1), (string)"string");
        Assert::AreEqual(patternClause.getArgType().at(2), (string)"assign");
}
```

Test purpose: Test if query with two clauses are valid

Required Test Inputs: Sample query strings

Expected Test Results: The argument and argument type should match the expected output

```
TEST_METHOD(testSyntaxError) {
        string queryString = "wr0ng";
        QueryValidator qv(queryString);
        QueryTable qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), true);

        queryString = "pattern such that Select";
        qv = QueryValidator(queryString);
        qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), true);

        queryString = "while w; assine a; assasin a; Serect BOOREAN";
        qv = QueryValidator(queryString);
        qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), true);

        queryString = "assign a1b2c3, aquaman, as0n; variable d4,vict0ry; while e5; constant
        c6; stmt salm0n3lla; Select c6 such that Modifies(salm0n3lla, _) puttern
        as0n(vict0ry,_\"on3l0ngstr1ng\"_)";
        qv = QueryValidator(queryString);
        qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), true);

        queryString = "assign abc; while w123; Select w123 such dat pattern abc(_,_)";
        qv = QueryValidator(queryString);
        qt = qv.parse();
        Assert::AreEqual(qt.isNullQuery(), true);
}
```

Test purpose: To test QueryValidator for syntax error
Required Test Inputs: Sample query strings
Expected Test Results: Assert that the result from isNullQuery() method matches the expected value

6. Discussion

(i) Rationale for the info duplication in the ModUsesTable - Having two tables containing the same information.

The `ModUsesbyStmtTable` store the variables modified and used by each line of the code, while the `ModUsesbyVarTable` being the inverse, maps each variable that appears in the program to the statement line where it is found.

The main reason for having two tables storing the exact same information, albeit the reverse use of key, is to allow a faster retrieval of inverse queries for modifies and uses. This can be illustrated using the two sample queries below:

1) Select v such that Modifies(1, v);
2) Select s such that Modifies(s, “x”)

In the first query; given a statement number, we return a list of variables that is modified by the statement. With the statement number, intuitively, it makes sense to search for the result through the `ModUsesbyStmtTable`.

In the second query; given a variable, we return a list containing all the statement numbers where the variable is modified. If there is only the `ModUsesbyStmtTable`, we would have to iterate through the uses table to search for variable x and subsequently for the corresponding statement numbers that are associated with variable x. This can affect the performance of the program i.e. time required to retrieve the required result, and we justify our decision to build the `ModUsesbyVarTable` as a similar table which maps a variable to where it is found, can easily resolve this issue.

(ii) Rationale for using static pointers for PKB and not mass instantiation
That is, why does the PKB implement a Singleton design pattern? Well, the PKB is a ‘state-ful’ class. It contains states, which has to be distributed to any component that requests a PKB. To ensure the states remain the same across every request, there has to be only one instance of the PKB class, at any one time. Thus a Singleton design pattern.

But a Singleton might not be the best solution. What we want is a single instance of a class. There are others who are wiser than us, who say that a single instance can be achieved with other means. They are against a Singleton implementation, because, in one of many reasons, the Singleton violates the Single Responsibility Principle. That is, the Singleton is in charge of policing class instantiations, and maintaining global information, when it should be fulfilling just one responsibility. More arguments against a Singleton pattern can be found in the article from year 2008, ‘Singleton I love you, but you're bringing me down’. URL: http://geekswithblogs.net/AngelEyes/archive/2013/09/08/singleton-i-love-you-but-youre-bringing-me-down-re-uploaded.aspx

Optimistically speaking, the code is in a state of flux, and a Singleton pattern is progress.

(iii) Rationale for not building an AST

As of iteration one, our team decided not to build an AST due to the issue of time constraints. At the same time, we believe that it is sufficient to store the relevant information parsed in individual tables i.e. ModUsesbyStmtTable, ModUsesbyVarTable, Parent, Follows and the Constant Table.

(iv) Internal Data structures Design Decisions

	Considerations	Pros	Cons	Decision
ModUsesTable (by statement), ParentTable, FollowsTable	**Use of unordered_map instead of vector to store key-pairs**	**O(1) retrieval O(1) insertion**	**O(lgn) insertion O(n) retrieval**	**Unordered_map is chosen over vector as we felt that searching will be done more frequently as compared to insertion and thus O(1) retrieval will be of higher priority as compared to O(1) insertion**

(v) Problems encountered

1. Took a longer time to come to a mutual consensus of the structure (implementation of the parser and PKB)
    a. Team members have varying opinions, for example some supported the idea of creating a statement object, whereas others felt that such abstractions were not necessary for Iteration 1.
2. Unfamiliarity with C++
    a. Time and effort needed to understand the basics and syntax of the language.
    b. Frequent debugging.

(vi) Areas to improve on

1. The team should work on starting their unit testing in the earlier project phase i.e. coding and unit testing should be done in parallel instead of starting on unit testing only after the code is done.
   It is always a good practice to start testing early to identify potential bugs or logic errors.

2. Team communication -- Clear all doubts before starting on the implementation.

7. Documentations of Abstract APIs

Parser component:

Tokenizer
Overview:
Tokenizer will read in a SIMPLE source file as input and convert it into individual string tokens for parsing

Public Interface:

void Tokenizer(**string** *fileName*);
Parameters:
    fileName must be a SIMPLE source file
Description:
    Constructor for Tokenizer object

string getNextToken();
Description:
    Returns the next token from the SIMPLE source file

Parser
Overview:
Parser accepts the user-specified SIMPLE source file and instantiates the Tokenizer to break the source code into string tokens. The parser iterates through the tokens while simultaneously populating the PKB.

Public Interface:

Parser Parser(**string** *fileName*);
Parameters:
    fileName must be a SIMPLE source file
Description:
    Constructor for Parser object

void process();
Description:
    Parses the SIMPLE source file which builds the PKB

PKB Component:

PKB
Overview:
PKB is a static class consisting of a PKB interface that provides APIs for the various data tables. It is a facade class that masks the internal structures of the PKB.

Public Interface:

static **PKB*** getPKB()
Description:
    Returns a static pointer to the PKB class, allowing access to the various table APIs

	void destroyInstance() Description: Dereferences all pointers to PKB and all its tables, effectively destroying the PKB	
	void addParent(**int** *lineOfParent*, **int** *lineNum*) **int** getParentOf(**int** *stmtNum*) **list<int>** getParentStar(**int** *stmtNum*) **list<int>** getChildrenOf(**int** *stmtNum*) **bool** isParentEmpty() **bool** isParentOf(**int** *parent*, **int** *child*) **bool** isParentStar(**int** *parent*, **int** *child*)	**Refer to ParentTable**
	void addFollows**(int** *stmt*, **int** *nestingIndex***);** **int** getFollowedFrom(**int** *stmtNum*) **int** getFollower(**int** *stmtNum*) **list<int>** getFollowedFromStar(**int** *stmtNum*) **list<int>** getFollowerStar(**int** *stmtNum*) **bool** isFollowEmpty() **bool** isValidFollows(**int** *s1*, **int** *s2*) **bool** isFollowsStar(**int** *s1*, **int** *s2*)	**Refer to FollowsTable**
	void addModifies(**int** *stmtNum*, **string** *var*) **void** addUses(**int** *stmtNum*, **string** *var*) **bool** isModified(**int** stmtNum, **string** *varName*) **bool** isUsed(**int** *stmtNum*, **string** *varName*) **list<string>** getModifiedBy(**int** *stmtNum*) **list <string>** getUsedBy(**int** *stmtNum*); **bool** isValidStmt(**int** *stmtNo*);	**Refer to ModUsesTablebyStmt**
	void addModifies(**string** *var*, **int** *stmtNum*) **void** addUses(**string** *var*, **int** *stmtNum*) **bool** isValidVar(**int** *var*) **list<int>** getModifiedBy(**string** *var*) **list <int>** getUsedBy(**string** *var*) **list <string>** getAllModVar() **list <string>** getAllUsedVar() **list <string>** getVarList()	**Refer to ModUsesTablebyVariable**
	void addStatement(**int** *stmtNum*, **string** *stmtType*) **list<int>** getWhileList() **list<int>** getAssignList() **list<int>** getStmtList() **int** getStatementCount()	**Refer to StatementTable**
	void addConstant(**int** *constant*, **int** *stmtline*) **list<int>** getConstantList(**void**) **list<int>** getStmtlineByConstant(**int** *constant*)	**Refer to ConstantTable**

FollowsTable
Overview:
FollowsTable stores associations between statement lines and nesting levels which are used to infer

Follows/Follows* relationships	
Public Interface:	
	void addFollows(**int** *stmtNum*, **int** *nestingLevel*) Description: Populates the FollowsTable with nesting level *nestingLevel* associated with *stmtNum*
	int getFollowedFrom(**int** *stmtNum*) Description: Returns the statement number of the statement that is followed from *stmtNum* Returns -1 if *stmtNum* has no statement followed from
	int getFollower(**int** *stmtNum*) Description: Returns the statement number of the statement following *stmtNum*. Returns -1 if *stmtNum* has no statement following it
	list<int> getFollowedFromStar(**int** *stmtNum*) Description: Returns a list of all statements that are followed from *stmtNum*, i.e. all s such that Follows*(s, *stmtNum*) Returns an empty list if *stmtNum* has no statement followed from
	list<int> getFollowerStar(**int** *stmtNum*) Description: Returns a list of all statements that follow *stmtNum*, i.e. all s such that Follows*(*stmtNum,* s) Returns an empty list if no statement follows *stmtNum*
	bool isFollowEmpty() Description: Returns true if there are no Follow/Follow* relationships in the table. Returns false otherwise.
	bool isValidFollows(**int** *s1*, **int** *s2*) Description: Returns true if Follows(s1, s2) holds.. Returns false otherwise.
	bool isFollowsStar(**int** *s1*, **int** *s2*) Description: Returns true if Follows*(s1, s2) holds. Returns false otherwise.

ParentTable *Overview:* ParentTable stores the parent association between statements which is used to infer Parent/Parent* relationships	
Public Interface:	
	void addParent(**int** *lineOfParent*, **int** *lineNum*) Description: Calls ParentTable.addParent(lineOfParent, lineNum) Populates the ParentTable with a Parent(*lineOfParent, lineNum)* relationship
	int getParentOf(**int** *stmtNum*) Description: Returns the parent of *stmtNum* Returns -1 if *stmtNum* has no parent
	list<int> getChildrenOf(**int** *stmtNum*) Description: Returns a list of all the child statements of *stmtNum* (i.e. all s for Parent(*stmtNum*, s)) Returns an empty list if *stmtNum* has no children
	bool isParentEmpty() Description: Returns true if there are no Parent/Parent* relationships in the table Returns false otherwise
	bool isParentOf(**int** *parentStatementNumber*, **int** *childStatementNumber*) Description: Returns true if the statement at *parentStmtNumber* is a parent of *childStmtNumber* Returns false otherwise
	list<int> getParentStar(**int** *stmtNum*) Description: Returns a list of all the transitive parent statements of *stmtNum* (i.e. all s for Parent*(*stmtNum*, s)) Returns an empty list if *stmtNum* has no parent
	bool isParentStar(**int** *parentStatementNumber*, **int** *childStatementNumber*) Description: Returns true if the statement at *parentStmtNumber* is a transitive parent of *childStmtNumber* Returns false otherwise

ModUsesTablebyVariable
Overview:
For each unique variable, ModUsesTablebyVariable stores the corresponding lines of code (statement number) that the variable has been modified or used by in the program.

Public Interface:

unordered_map<string, list<int>> getModTable()
Description:
Returns a table containing a mapping of unique variables to corresponding lists of statements numbers.

unordered_map<string, list<int>> getUsesTable()
Description:
Returns a table containing a mapping of unique statement variables to corresponding lists of statement numbers.

void addModifies(**string** *var*, **int** *stmtNum*)
Description:
Insert statement number *stmtNum* corresponding to the modified variable *var* into the table.

void addUses(**string** *var*, **int** *stmtNum*)
Description:
Insert statement number *stmtNum* corresponding to the variable *var* used into the table.

bool isValidVar(**int** *var*)
Description:
Returns true if variable *var* is found in the table

Return false otherwise

list<int> getModifiedBy(**string** *var*)
Description:
Returns a list of statement numbers where variable *var* has been modified

Returns the empty list if *var* has not been modified by any statement

list <int> getUsedBy(**string** *var*)
Description:
Returns a list of statement numbers where variable *var* has been used

Returns the empty list if *var* has not been used in any statement

list <string> getAllModVar()
Description:
Returns a list of all the variables modified

list <string> getAllUsedVar()
Description:
Returns a list of all the variables used

list <string> getVarList()
Description:

	Returns a list of all the variables in the program

ModUsesTablebyStmt *Overview:* ModUsesTablebyStmt stores the variables that are modified and used by each line of code in the given program.	
Public Interface:	
	unordered_map<int, list<string>> getModTable() Description: Returns a table containing a mapping of unique statement numbers to corresponding lists of variables modified.
	unordered_map<int, list<string>> getUsesTable() Description: Returns a table containing a mapping of unique statement numbers to corresponding lists of variables used.
	void addModifies(**int** *stmtNum*, **string** *var*) Description: Insert variable *var* corresponding to statement number *stmtNum* into the modifesTable
	void addUses(**int** *stmtNum*, **string** *var*) Description: Insert variable *var* corresponding to statement number *stmtNum* into the usesTable
	bool isModified(**int** *stmtNum*, **string** *varName*) Description: Returns true if the statement at *stmtNum* modifies the variable *varName* Returns false otherwise
	bool isUsed(**int** *stmtNum*, **string** *varName*) Description: Returns true if the statement at *stmtNum* uses the variable *varName*
	list<string> getModifiedBy(int *stmtNum*) Description: Return the list of variables modified by statement number *stmtNum*
	list <string> getUsedBy(int *stmtNum*) Description: Return the list of variables used by statement number *stmtNum*
	bool isValidStmt**(int** *stmtNo***)** Description: Return true if statement number *stmtNo* is found in the table. Returns false otherwise.

StatementTable *Overview:* Stores associations between statement line and statement type ("while" or "assign")	
Public Interface:	
	void addStatement(**int** *stmtNum*, **string** *stmtType*) Description: Insert statement number *stmtNum* of statement type *stmtType* into the table.
	list<int> getWhileList() Description: Returns a list of statement lines containing while statements.
	list<int> getAssignList() Description: Returns a list of statement lines containing assignment statements.
	list<int> getStmtList() Description: Returns a list of all the statement lines in the program.
	int getStatementCount() Description: Returns the number of statements in the program.

ConstantTable *Overview:* ConstantTable contains a list of all the constants and a map with a constant as the key and a list of statement lines which uses the constant as value	
Public Interface:	
	void addConstant(**int** *constant*, **int** *stmtline*) Description: Insert constant *constant* used by statement number *stmtline* into the constant table
	list<int> getConstantList() Returns a list of all the constants
	list<int> getStmtlineByConstant(**int** *constant*) Returns a list of statement lines which contains constant *constant*

PQL Component:

QueryResultProjector
Overview:
QueryResultProjector is responsible for formatting the results received from QueryEvaluator and outputting the result to the caller with getResults() method

Public Interface:

QueryResultProjector QueryResultProjector(**QueryTable** *input*)
Parameters:
Accepts a QueryTable structure with Clause objects initialized with the results of the evaluation of the query
Description:
Overloaded constructor method for QueryResultProjector

list<string> getResults()
Description:
Returns the results from the evaluated query in a list. Returns an empty list if there are no results for the query

bool isResultShareCommonSyn(**string** *selectSyn*, **QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*)
Parameters:
Accepts a string synonym and two QueryResult pointers
Description:
Returns TRUE if result is common synonym

vector<string> getClauseSynonym(**string** *clause*, **QueryResult***)
Parameters:
Accepts a string clause and a QueryResult pointer
Description:
Return a list of clause synonyms

list<string> mergeResult(**QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*)
Parameters:
Accepts two QueryResult pointer
Description:
Return a list of merge results of *suchThatResult* and *patternResult*

list<string> mergeResult(**string** *selectSyn*, **QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*)
Parameters:
Accepts a string clause and two QueryResult pointers
Description:
Return a list of merge results of *suchThatResult* and *patternResult*

list<string> getListIntersection(**list<string>** *result1*, **list<string>** *result2*)
Parameters:
Accepts two list of string
Description:
Return a list of string:

	list<string> getListResult(**vector<string>** *vectorResults*) Parameters: Accepts a vector of string Description: Return a list of string::
	list<string> getSynResult(**string** *syn*, **string** *clause*, **QueryResult*** *clauseResult*) Parameters: Accepts a string and QueryResult pointer Description: Return a list of string
	list<string> getCommonSynonym(**QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*) Parameters: Accepts a string and QueryResult pointer Description: Return a list of string
	unordered_map<string, list<string>> getCommonSynonymResult(**QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*) Parameters: Accepts two QueryResult pointers Description: Return an unordered_map
	unordered_map<string, list<string>> getCommonSynonymResult(**list<string>** *commonSyn*, **QueryResult*** *suchThatResult*, **QueryResult*** *patternResult*) Parameters: Accepts a list of QueryResult pointer two QueryResult pointers Description: Return an unordered_map

QueryEvaluator *Overview:* QueryEvaluator is in charge of query evaluation. Passes QueryResultProjector the results of the evaluate() method.	
Public Interface:	
	QueryEvaluator QueryEvaluator(**QueryTable**) Parameters: Accepts a QueryTable structure Description: Overloaded constructor method for QueryEvaluator
	QueryTable* evaluate() Description: Returns a pointer to a QueryTable that contains the results of the evaluated query

QueryValidator *Overview:* QueryValidator is responsible for parsing user-queries. Passes QueryEvaluator the details of the query with the parse() method	
Public Interface:	
	int getTokenType() Description: Returns an integer
	string getTokenName() Description:

Clause *Overview:* Abstract class representing a condition in the user-specified query	
Public Interface:	
	Clause() Description: Default constructor method for Clause that initializes a Clause object with no parameters
	Clause(**string** *relation*, **vector<string>** *arg*, **vector<string>** *argType*); Parameters: Accept a **string** relation that indicates the relationship of this clause, a **vector<string>** *arg* containing the arguments to the clause, and a **vector<string>** *argType*. Description: Overloaded constructor method for QueryEvaluator
	string getRelation() Description: Returns the relationship of this clause
	vector<string> getVar() Description: Returns a **vector<string>** that contains the arguments to this clause, where the first and second elements are the first and second arguments to the clause respectively
	vector<string> getVarType() Description: Returns a **vector<string>** that contains the argument types of the arguments to this clause, where the first and second elements are the first and second argument types to the arguments of the clause respectively

QueryTable *Overview:* Maintains a table of clauses derived from the user-specified query, as well as their validity and results	
Public Interface:	
	QueryTable(**Clause*** *selectClause*, **Clause*** *suchThatClause*, **Clause*** *patternClause*) Parameters: Accepts three Clause pointers for the for the select, such that and pattern clause. If the clause is not present in the query, the pointer will be null. Description: Overloaded constructor method for QueryTable
	Clause* getSelectClause() Description: Returns a pointer to the **Clause** object that represents the select clause
	Clause* getSuchThatClause() Description: Returns a pointer to the **Clause** object that represents the such that clause
	Clause* getPatternClause() Description: Returns a pointer to the **Clause** object that represents the pattern clause
	Clause* setSelectClause(**Clause*** *selectClause*) Parameters: Accepts a pointer to a Clause object Description: Sets the pointer for the ‘select’ clause to the input pointer
	Clause* setSuchThatClause(**Clause*** *suchThatClause*) Parameters: Accepts a pointer to a Clause object and Description: Sets the pointer for the ‘such that’ clause to the input pointer.
	Clause* setPatternClause(**Clause*** *patternClause*) Parameters: Accepts a pointer to a Clause object Description: Sets the pointer for the ‘pattern’ clause to the input pointer
	bool isSuchThatAvail() Description: Returns **TRUE** if the QueryResult pointer for the ‘select’ clause results is not null. Returns FALSE otherwise.
	bool isPatternAvail() Description: Returns a pointer to the **Clause** object that represents the select clause

	bool isSelectResultEmpty() Description: Returns a pointer to the **Clause** object that represents the select clause
	bool isSuchThatResultEmpty() Description: Returns a pointer to the **Clause** object that represents the select clause
	bool isPaternResultEmpty() Description: Returns a pointer to the **Clause** object that represents the select clause