CS3203 Test 2 (2020/21 Sem 1)

**[Section II]**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name: Soh Yi Zhi Zeke** | | **Student No.: A0190203W** | **Group No.:**  **05A** |
| **Question answered:**  **1** | **Marks Awarded: (For Examiner’s Use Only)** | | |

**Duration: 30 minutes Max Marks: 30**

1. Answer only ONE (1) question in this section on your own. You can discuss with your groupmates. However, you are NOT allowed to answer the same question as any of your groupmates.
2. You can type / handwrite your answers on a hardcopy / softcopy of the question paper or other sheets of paper.
3. **Create a PDF file** containing **your answers** and **the required information as shown in the box above**.
   1. **Name your PDF** using your student number (e.g. A1234567Z.pdf)
   2. Ensure that there are **FIFTEEN (15) pages or less** in your PDF. Any answers written beyond page 15 will **NOT** be marked. You can consider deleting pages containing the questions.
   3. Ensure that there are **NO landscape pages** in your PDF. If you wish to include diagrams in landscape mode, rotate the diagram clockwise by 90 degrees.
   4. Ensure that your answers are legible when printed on A4 paper.
4. **Upload the PDF** by **27 Oct (Tue), 5:25pm** on LumiNUS Files 🡪 Student Submission 🡪 Test 2 Section II Question *X*, where *X* is the question number of the question you answered.
   1. Failing to submit your answers to the correct LumiNUS folder by the stated deadline means you will get **0** marks for Section II. Late submissions are **NOT** allowed.
   2. Only the most recent submission will be graded.
5. Address any questions you might have to the host of the Zoom session.
6. Concise, point-wise answers are accepted and preferred over long-paragraph answers.
7. it is **NOT** compulsory for you to describe the design of your own SPA system and the components mentioned in the headers of the questions are only indicative. You may choose to describe any reasonable design and it does not have to match with the components mentioned.

**For Question 1, refer to Fig. 1.**

*Figure 1: Program for call validation*

procedure main {

1. call a;
2. call d; }

procedure a {

1. call b; }

procedure b {

1. call main; }

procedure b {

1. call b; }

**For Questions 2 and 3, refer to Fig. 2.**

*Figure 2: Program for parent and call relationships*

procedure main {

1. while (a < 1) {
2. if (c == 0) then {
3. a = a + b;

} else {

1. while (d > 1) {
2. d = a \* 1;
3. call procA; }}
4. call procC; }}

procedure procA {

1. call procB;
2. b = a + 1;
3. call procC; }

procedure procB {

1. call procC; }

procedure procC {

1. a = 1; }

**For Questions 5 and 6 refer Fig. 3.**

*Figure 3: Pattern extension*

Consider the extension of pattern for if statement such that sub expressions are allowed in the placeholder blocks. For a general if pattern clause **pattern ifs (con\_expr, then\_expr, else\_expr)**

The new rules are as follows for any if statement:

a) *then\_expr* and *else\_expr* can be any sub expression or exact expression or place holders.

b) the statement is true if there exists an assignment statement in the corresponding then and / or else block of the if statement such that **pattern a (\_, then\_expr)** or **pattern a (\_, else\_expr)** is true.

c) both the if and assignment statement must be in the same procedure.

For the following program and patterns, the numbers in the square brackets are the results for the ifs synonym.

1) pattern ifs (\_, "a + b", \_) [1]

2) pattern ifs (\_, \_, "a \* b") [1, 4]

3) pattern ifs (\_, "a \* b", \_) [None]

4) pattern ifs (\_, "c", \_"b"\_) [4]

procedure main {

1. if (a < 1) then {

2. while (a < 1) {

3. a = a + b; }

} else {

4. if (a == 0) then {

5. a = c;

} else {

6. a = a \* b; }}}

# **SPA Front-end / Validation**

1. **[12 marks]** Explain how to validate semantic errors related to **call** statements. Your explanation should cover (but not limited to) the semantic errors (e.g., cyclic calls) in the source program in Fig. 1.

Focus your answer on the specific errors (instead of giving the general algorithm for validation) and discuss each error separately (instead of saying the validation will stop after the first error).

1. **[18 marks]** Explain how to syntactically validate the right-hand-side expressions in **assign** statements. Give 2 examples of possible violations of syntax rules related to such expressions and discuss how these violations are identified. You can assume that all the constants and variable names in the expressions are valid.

Note: It is possible for a design to perform certain validations implicitly as part of the parsing process (e.g., the source program is free of certain errors if the parsing can be completed for the whole program). If you want to describe such a design in your answer, give details about the parsing process and highlight how the parsing would stop halfway due to certain errors.

1a.

Possible semantic errors for call statements would be the **cyclic calls** as well as the **calling of procedures that do not exist** within the input source program.

During the parsing of the source program, a list of procedure names are recorded for each procedure statement encountered (for the remaining explanation, it will be regarded as *procedures*) and a list maintains the procedure name in each call statement (this will be referred to as *calls)*. To perform validation for the calling of procedures that do not exist within the input source program, after parsing the entire program once, we have a list of procedure names in calls. We verify the procname of each call statements in *calls* against the procedure names stored in *procedures.* The time complexity would be O(m x n).

Step1:

During the parseString method, a set of procedure nodes, a Set<PNodes> is maintained. This set is updated at each procedure and call statement that is parsed. Additionally, a PNode, currNode, is maintained to track the current procedure being parsed. A summary of the maintained variables can be found in the table below and will be referenced from here on in italics.

|  |  |
| --- | --- |
| **Object type** | **Variable name** |
| *Set<PNode>* | *procedures* |
| *PNode* | *currNode* |

Table 1a: Variables maintained for building of directed graph

Upon parsing a procedure statement, we first check if the set already contains a *PNode* with the same name as the encountered procedure. This check is necessary as the procedure may have been called by an earlier procedure. If a *PNode* with the procedure name can be found within *procedures*, the *currNode* is set to that. If none of the *PNodes*’ procedure names match the encountered procedure, we create a new *PNode*, add it to *procedures* and assign it as the *currNode*.

When parsing call statements, we once again check if *procedures* contain a *PNode* with the procedure name provided in the call statement. If such a *PNode* exists, we add it as an edge to *currNode*. Else, a new *PNode* is created and added as an edge to *currNode*. We then add the new *PNode* into *procedures.*

Step 2: Obtaining reverse topologically sorted procedure call order

After we have parsed all the statements, we should have a complete directed graph of the sample source. We then perform a topological sort on the directed graph from step 1. This is done simply by doing DFS and maintaining a stack. As we need to obtain a sequence of procedures from the **last** procedure that is called to the **first** procedure, the results from the stack obtained in the topological sorting needs to be **reversed.**

Finally cyclic detection is done of the attained reverse topological ordering and if a cycle is detected, an exception will be thrown indicating that a cycle is detected.

**1b.Validation of RHS of assignment statements**

1. First, we check if the given string is empty and return false if so. We then check if the given string is IsName or IsConst. If either of them is true, immediately return true.
2. Using the following code as a positive example: RHS = “(4 + a)”

(4 + 5) is neither NAME or CONST in SIMPLE syntax and returns false.

|  |
| --- |
| **Parser REGEX 2** |
| -+\*/%( |

Table 1c: Regex to find operator character

2a. Using regex search for the above regex to identify the first occurrence of any of the above regex and its position. If ( is found, to find the matching ) bracket, we maintain a stack while iterating through the given string in a for loop. Any ( encountered is pushed as an instance into the stack. Upon encountering a ) character, the stack is then popped.  In the example, the position of the matching ) is at index **6**.

One possible syntax violation would be if the bracketings do not match, and the Validator should immediately return false and throw an exception which is resolved in the implementation of step 2a.

* 1. The given string is then divided from the **given position** to form a LHS and RHS. In the example string, we have LHS = “(4 + a)” and RHS = “”. If the RHS is empty, the enclosing () is removed from the LHS and we repeat the process from step 1 recursively calling on the remaining string, “4 + a”.

2b. In the event any other of the above regex is matched, the string would then be divided into LHS and RHS of the position of the match. In the example we would have 4 + a being split into LHS = 4 and RHS = a and it would be valid.

Another possible syntax violation would be the presence of space between variables without an operator. For example the RHS now becomes “4 + a 5”

Following step 1 to step 2b, we would have LHS = “4” and RHS = “a 5”. we would then validate if “a 5” has a space character within it. If so we have identified the syntactic violation and RHS = “a 5” will fail validation and a exception will be thrown.

# To ensure that cyclic/recursive calls are not present in the source code, we perform cyclic detection on the obtained reverse topological ordering. Upon detection of a cycle in the directed graph, the **Parser** will throw an error message and exit the program. **Front-end / PKB population**

* 1. **[18 marks]** Given the program in Fig. 2 and assuming that the Front-end populates the **Parent** and **Parent\*** relationships, describe the complete sequence of steps involved in setting such relationships for procedure **main**.

Use API calls in your description (e.g., setParent(1,2)), briefly describe these APIs, and indicate which component in the Front-end (e.g., Parser, Design Extractor) makes these calls.

Focus on explaining based on the code example provided. Do **NOT** attempt to provide a general explanation.

* 1. **[12 marks]** Describe an alternative method of populating **Parent\*** relationship that is substantially different from your described approach in part a). Critically compare your current approach with this alternate approach using a list of criteria.

Note: If you want to describe your actual design but it does not follow the assumption above, you can describe it as the alternative method instead.

# **PKB**

Provide a clear **description of the data structures** used to store the **Calls and Calls\*** relationships for statements in your PKB. In your answer, describe the following:

1. **[15 marks]** Describe the data structures in your PKB for storing the Calls and Calls\* relationships. Describe the content of your data structures based on the program given in Fig. 2.
2. **[15 marks]** Describe the APIs provided to the Front-end (Parser, Design Extractor) and Query Evaluator for populating and retrieving these relationships. Explain in what context each API operation would be used. For APIs provided to the Query Evaluator, explain each API with a sample query.

Note: If you want to describe your actual design but the APIs mentioned do not exist (or exist as one single, possibly trivial API) on PKB for the Front-end and / or the PQL, provide a description on how the population and / or retrieval is achieved. You should still include similar types of information and sample queries as required by the question.

# **Query Preprocessor / Query Validation**

Describe a possible design for query validation. Avoid giving a general algorithm for validating queries, but rather give specific answers applying to the given queries.

* 1. **[12 marks]** Describe the data structure(s) involved in detecting the issues in queries (i)-(iii) as mentioned below.
  2. **[9 marks]** Explain the steps involved in validating queries (i)-(iii) and explain how the data structures described in part a) are used.

1. **assign a; variable v;**

**Select s such that Uses (a, "v") and Modifies (a, v)**

1. **stmt s; assign a; print p; procedure p;**

**Select s such that Uses (a, "v") and Modifies (p, v**)

1. **assign a;**

**Select a such that Uses (a, 1)**

In particular, queries (i)-(iii) contain the following errors: **undeclared synonym** (i. and ii.), **ambiguous synonym** (ii.) **and incorrect argument type** (iii).

* 1. **[9 marks]** Describe how type-validation can be done for the **such that** clauses in queries (iv)-(vii). Your explanation should cover all the relationships shown in those queries:

1. **assign a; prog\_line n;**

**Select a such that Next\* (a, n)**

1. **assign a; if ifs;**

**Select a such that Parent\* (ifs, a)**

1. **variable v;**

**Select v such that Modifies (12, v)**

1. **procedure p;**

**Select p such that Calls (p, "a")**

Note: It is possible for a design to perform certain validations implicitly as part of the parsing process (e.g., the query is free of certain errors if the parsing can be completed for the whole query). If you want to describe such a design in your answer, give details about the parsing process and highlight how the parsing would stop halfway due to certain errors.

# **Query Evaluator / PKB**

1. **[12 marks]** Describe the process of evaluating **pattern** clauses using the query given below. Explain how the evaluator retrieves the results for each of the pattern clauses. Include at least the following information:
2. Data structures used to store the pattern clauses.
3. API call(s) used to retrieve results for pattern clauses.
4. Name the components that provide the API calls.

You do **NOT** need to explain how the pattern clause is parsed and how the results for each clause are filtered and merged.

**assign a1, a2; if ifs; while w; variable v;**

**Select <a1, w> pattern a1 (\_, "a + b") and a2 (v, \_"b"\_) and a3 ("y", "c + d") and ifs (v, \_, \_) and w (\_, \_)**

1. **[18 marks]** Consider the extension to **pattern** clause as described in Fig. 3 that allows **pattern ifs ("x", \_"a \* b"\_, "a")** to be accepted and evaluated.

Describe changes required in your current data structures and query evaluation method. Mention clearly the changes required in PKB (e.g. additional information to be stored and changes in API) and/or in Query Evaluator (e.g. evaluation algorithm and result structure).

# **Testing**

1. **[12 marks]** How do you design **a set of SIMPLE source programs** for testing queries involving *pattern* relationship for the full SPA requirements (i.e., **NOT** including the extension described in Fig. 3)? Describe your strategies for creating the source programs and provide a few samples to illustrate your design.
2. **[18 marks]** How do you design **a set of queries** for testing the *pattern* relationship for the full SPA requirements + the extension described in Fig. 3? Describe your strategies for creating such queries and provide a few samples to illustrate your design.

The following pages are left blank intentionally. Please write your answer from this page onwards.