University of Toronto
Faculty of Arts and Science

April 2016 Examinations

CSC488H1S / CSC2107HS

Duration – 2 hours (120 minutes)

OPEN BOOK ALL written aids, books and notes are allowed.
ALL non-programmable calculators allowed.
NO other electronic aids allowed.

120 marks total, 8 Questions on 5 Pages. ANSWER ALL QUESTIONS
Write all answers in the Exam book.

You must receive a mark of 35% or greater on this final exam to pass the course.

WRITE LEGIBLY Unreadable answers cannot be marked.
Line/rule reference numbers on the left side of of programs and grammars are provided for ease of reference only and are not part of the program or grammar .
The notation **...** stands for correct code that has been omitted for brevity
State clearly any assumptions that you have to make to answer a question.

1. [15 marks] A proposal has been made to add array assignments to the project language.
Add syntax:

statement: arrayname ':' '=' arrayname

For example:

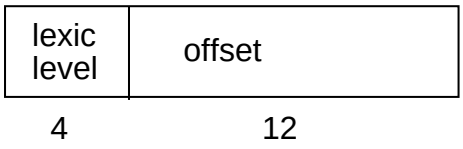
```
 1    var A[ 1 .. 10 ] , B[ 1 .. 10 ] : integer
 2    var P[ -10 .. 10 , 17 ] , Q[ -10 .. 10 , 17 ] : boolean
      ...
14    A := B
15    Q := P
```

a) Design a set of semantic analysis operations for checking the semantic correctness of array assignments.
b) Design a code generation template for the general cases of one and two dimensional array assignment.

2. [15 marks] The hardware designer of the pseudo machine used in the course project is working on version 2.0 of the pseudo machine.

One proposed improvement to allow larger programs in the 16,384 words of memory is to change the encoding of the ADDR instruction.

Instead of using one memory word for the lexical level and one word for the offset, the lexical level and offset would be encoded in one 16-bit word:



using 4 bits for the lexical level and 12 bits for the offset.

Describe the changes that would be required in the project compiler to implement this new instruction format.

3. [20 marks] Consider the program fragments in a Pascal-like language below:

```
 1    var a array 1 .. 20 , -15 .. 15 of float
 2    var b array 1 .. 400 of float
 3    var J , K : integer
      … % Assume variables are initialized here
20    for J := 1 to 20 do
21        a[ J , J ] := 0
22        for K := -15 to 15 do
23            b[ 16 * J + K - 15 ] := a[ J , K ]
24            a[ J , K ] := a[ J , K ]**2 - 3.0 * a[ J , K ] + 1.0
25        end for
26    end for
```

Where ** is the exponentiation operator, and float variables are stored in 4 bytes or memory.

Describe classical optimizations that a good optimizing compiler would perform on the code in lines 20 .. 26. You do **not** have to show every step in the optimization separately. You can show only the final result as long as it's clear what optimizations have been performed.

4. [5 marks] In lectures the instructor described a method for optimizing array subscription by folding known constant information about the array into the array address to avoid unnecessary runtime normalizing of array subscripts. For example:

Given **var** B [lb_1 .. ub_1 , lb_2 .. ub_2] **integer**

Subscript Code	B[$expr_1$, $expr_2$]	
1	ADDR	$LL_B \quad OFFSET_B - ((ub_1 - lb_1 + 1) * lb_1 + lb_2)$
2	PUSH	$ub_1 - lb_1 + 1$
3	$expr_1$	
4	MUL	
5	$expr_2$	
6	ADD	
7	LOAD	

Are there any circumstances where doing this improvement could lead to incorrect results?

5. [10 marks] Many programming languages allow *block comments* For example:

```
/*************************************************************
 *   Start of long comments
 *   long comment continues
 *   for a very long while
 *   and may include parts of comment marker like  * or /
 *   block comment ends on next line
 *************************************************************/
```

Describe how a lexical analyzer could *efficiently* process block comments.
Assume that block comments begin with `/*` and end with `*/`.

6. [15 marks] Assume the code fragment below in the course project language:

```
1     var P, Q, R : boolean

      …     % variables given values here

10    R := ( P or not Q ? not R or not P ? not ( P and ( Q or not R) ))
```

Show a translation of the statement on line 10 into the *quadruples* used in lectures to describe translation.

7. [20 marks] List the semantic analysis checks that a Java compiler would perform on the Java code fragment shown below

```
1     public static void shell( int[] a ) {
2         int increment = a.length / 2;
3         while ( increment > 0 ) {
4             for ( int i = increment; i < a.length; i++ ) {
5                 int k = i;
6                 int temp = a[ i ];
7                 while ( k >= increment && a[ k - increment ] > temp ) {
8                     a[ k ] = a[ k - increment ];
9                     k = k - increment;
10                }
11                a[ k ] = temp;
12            }
13            if ( increment == 2 ) {
14                increment = 1;
15            } else {
16                increment *= (5.0 / 11);
17            }
18        }
19    }
```

8.[20 marks] In the programming language Modula-3 the definition of a procedure has the syntax:

```
1   ProcedureDeclaration:   ProcedureHead  '=' Block Identifier
2   ProcedureHead:          'PROCEDURE' Identifier Signature
3   Signature:              '(' FormalParameters ')' optType
4   optType:                ':' Type ,      % function
5                           % empty       procedure
7   FormalParameters:       FormalParameterList ,
8                           % empty
9   FormalParameterList:    FormalParameter ,
10                          FormalParameterList ';' FormalParameter
11  FormalParameter:        AccessType IdentifierList ':' Type optInitializer
12  AccessType:             'VALUE' ,
13                          'VAR' ,
14                          'READONLY' .
15                          % empty
16  optInitializer:         ':' '=' constantExpression
17                          % empty
```

Where terminal symbols are enclosed in single quotes ('). Identifier is also a terminal symbol.

Write a recursive descent parser function

boolean function procedureDeclaration( )

that recognizes procedure declarations in Modula-3.

You may assume:
– your recognizer is called when the declaration parser recognizes the reserved word PROCEDURE
– there is a function **boolean function** Type ( ) that recognizes all types
– there is a function **boolean function** IdentifierList( ) that recognizes lists of identifiers
– there is a function **boolean function** Block( ) that recognizes the body of a procedure
– there is a function **boolean function** constantExpression( ) that recognizes all forms of constant expression
– there is a function **boolean function** getToken( ) that advances the scanner input to the next token
– all of these functions return **true** if successful and **false** otherwise
– the global variable *nextToken* contains the next input token from the scanner
– you may use auxiliary parsing functions if you wish

Total Marks = 120