University of Toronto

Faculty of Arts and Science

April/May Examinations 2000

CSC488S CSC2107S

Duration - 2 hours - 120 minutes

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

Part I - 4 Questions 80 marks, Part II - 4 Questions 40 marks. 120 marks total.
You must receive a mark of 35% or greater on this exam to pass the course.

WRITE LEGIBLY. Unreadable answers are not answers.
State clearly any assumptions that you have to make to answer a question.

Conventions for all questions:

In grammars, uppercase letters are nonterminal symbols and lower case letters are terminal symbols. λ is the empty string.
Line/rule reference numbers on the left side 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 clarity.

Part I - Answer ANY 4 Questions in Part I

IF MORE THAN 4 QUESTIONS ARE ANSWERED, ONLY THE FIRST 4 WILL BE COUNTED.

1. [20 marks] Describe the optimizations that a good optimizing compiler would perform on the C function listed below. You may show only the final result of the optimization **if** you very clearly describe all of the optimizations that have been performed. Assume that float variables require 4 bytes of storage.

```
 1    void batchersort( float A[] , int L , int R ){
 2        int I, J, K, P, N = R - L + 1 ;
 3        float T ;
 4        for( P = 1 ; P < N ; P += P )
 5            for( K = P ; K > 0 ; K /= 2 )
 6                for( J = K % P ; J + K < N ; J += (K + K))
 7                    for( I = 0 ; I < K ; i++ )
 8                        if( ( J + I + K ) < N )
 9                            if( A[ L+ J + I ] < A[ L+ J + I + K ]){
10                                T = A[ L+ J + I ] ;
11                                A[ L+ J + I ] = A[ L+ J + I + K ] ;
12                                A[ L+ J + I + K ] = T ;
13                            }
14        return;
15    }
```

2. [20 marks] In C++ and Modula-3 there is an exception handling mechanism based on the **try** and **catch** mechanism. An example (in C++ syntax) is given below.

```
 1      void testForBorg() {
 2          ...
 3          throw  "Borg Sighted" ;
 4          ...
 5      }
 6      ...
 7      try   {
 8          ...
 9          testForBorg() ;
10          ...
11          if( badness > 1 )
12              throw  23 ;
13          ...
14      }
15      catch( int  code ) { ... }
16      catch( char * msg ){ ... }
17      ...
```

The try block associates some set of statements (including function calls) with a set of catch handlers. The throw statement signals that an exception has occurred. Note that exception signaling is not limited to statements in the try block, an exception can also be signaled by a function called from the try block (e.g. testForBorg). In C++ the decision on which catch handler to invoke is based on the *type* of value used in the throw statement. catch handlers can have any type of formal parameter that is legal for a function in C++. In the example above the throw on line 3 will be caught by the handler on line 16 and the throw on line 12 will be caught by the handler on line 15. try blocks may be nested. An exception will propagate outward through nested try blocks until a catch handler with a parameter type that matches the type of the thrown exception is found.

Design an implementation for this form of try/catch exceptions. Describe how your implementation will deal with

- storage management, unwinding the run time stack.
- keeping track of active catch handlers.
- associating an exception signaled by a throw statement with the nearest (dynamically) enclosing catch handler of the correct type.
- show the code that you will generate for the throw, try and catch constructs in the example above.

3. [20 marks] The lecture notes described a method for implementing **case** or **switch** statements where the set of case labels was compact and a branch table implementation was suitable.

   Design a *complete* similar mechanism for implementing **case** or **switch** statements for the situation where the set of case labels is spare and non-compact (e.g. 1, 100, 10000, 10000000) and an if statement-like implementation is appropriate. Show the translations steps for all parts of the case/switch statement as was done in the lecture slides. Show the branching code that your would generate and describe the compile time data structures that you would need to do the code generation.

4. [20 marks] Transform the grammar below into LL(1) form:

1	S	→	S	a	T		
2		→	T				
3	T	→	A	b	B		
4	A	→	A	c	x		
5		→	x				
6	B	→	D				
7		→	d	e	E	f	g B
8	D	→	x				
9		→	F	h	F		
10	F	→	G	y			
11		→	x				
12	G	→	p				
13		→	m				
14		→	λ				
15	E	→	E	c	D		
16		→	D				

5. [20 marks] A proposal has been made to extend the language used in the course project by adding a **with** statement similar to the with statement in Pascal. The statement would have the definition:

withStmt	:	**with** withList **do** statement			
withList	:	oneWith	,		
		withList	','	oneWith	
oneWith		:	variable	'as'	identifier

   The purpose of this statement is to provide efficient access to the variables in the withList in the body of the statement. An example:

```
1    with A[J] as X , A[K+J] as Y do
2        % use X and Y here instead of A[J] and A[K+J]
```

   Describe the static semantic analysis that should be applied to this statement.
   Describe the code that would be generated for this statement on the machine used for the course project.

Part II - Answer ANY 4 Questions in Part II

IF MORE THAN 4 QUESTIONS ARE ANSWERED, ONLY THE FIRST 4 WILL BE COUNTED.

6. [10 marks] Consider a formatted print statement of the form:

print *format-string expression-list*

Where *format-string* is a string constant which defines the layout of the printed output. A format string may contain the special markers:

%d insert an integer expression at this point
%s insert a string expression at this point

Any other characters occurring in *format-string* are passed as is to the output. The *expression-list* is a list of values that are to be printed. Execution of this statement starts at the beginning of the *format-string* and the beginning of the *expression-list*. Ordinary characters in the *format-string* are simply copied into the print output buffer. If a special marker (%d or %s) is encountered, the next value from the *expression-list* is converted into its character representation and copied into the output buffer. There must be exactly as many special markers in the *format-string* as there are expressions in the *expression-list*.

Describe how you would implement this statement in the general case. What static semantic checks would you perform during compilation? Show the code (tuples) that you would generate for the typical print statement:

print "Hello World, Today is %s the %d day of %s in %d",
dayName, dayOrdinal , monthName, year

7. [10 marks] A proposal has been make to extend C to allow initial values for struct fields in typedef declarations. Every variable created using such a type would be initialized by the compiler to the values specified in the typedef declaration. For example

```
1    typedef
2        struct studentStruct {
3            char * name = "Anonymous" ;
4            long   number = 0999999999  ;
5            int    projectTeam ;
6            short marks[6] = { 0, 0 , 0, 0, 0 , 0 } ;
7            short  rawMark ;
8        } studentRecord ;
```

Discuss how you would implement this language feature. What implementation problems/issues are raised by this proposal? Discuss the impact that this feature would have on the compilers symbol and type table mechanisms, semantic analysis, run time storage organization and code generation.

8. [10 marks] Show how the Boolean expression given below would be translated into branching code.

W | X <= Y & Z != 0 | ! (P&Q&R) & Z > X

9. [10 marks] Describe how the structure declaration shown below would be mapped into memory using the space conserving Algorithm 2 described in the lecture notes.

```
 1      struct mapMeIfYouCan {
 2          char key   ;
 3          double     hashKey ;
 4          struct {
 5              short  index ;
 6              double oldKey  ;
 7              int *  locator ;
 8          } info ;
 9          union  {
10              short  keyCode ;
11              long masterCode ;
12          } codeInfo ;
13      } ;
```

Assume the following size and alignment constraints (in bits)

Type	Size	Align	Type	Size	Align
char	8	8	pointer	32	32
int	32	32	short	16	16
double	64	64	long	32	32

10. [10 marks] In Pascal, value and reference parameter modes are often confused. In particular, because value mode is the default, value parameters are sometimes used in a manner that suggests that reference mode was intended. A sign of this problem is an assignment to a value parameter before the value of the parameter has been used. Show how data flow analysis could be used to identify value parameters that are defined (i.e. assigned to) before they are used.

11. [10 marks] You have been as a compiler guru by Microsoft. Their operating system software has become so large (e.g. 50,000,000 lines of C code) that compiling it is very time consuming. They want fast compilation of extremely large programs with a high level of optimization.

Describe the design of a compiler that can efficiently process extremely large programs. Discuss the organization of the compiler and the major data structures that the compiler will use.

12. [10 marks] Assume a display is used to address the activation records of procedures and functions as was described in lecture. For the procedures declared below, show the state of the display after the sequence of procedure calls

$$P \rightarrow Q \rightarrow Q \rightarrow Q \rightarrow A(R) \text{ -> } F$$

Show both the static and dynamic links. Which activation records are addressable when F has been called in this sequence?

```
 1    procedure P( ... )
 2        procedure R( ... )
 3            ...
 4        end R
 5        procedure Q( ... )
 6            ...
 7            A( R );
 8            ...
 9        end Q
10        ...
11        Q( .. )
12    end P
13    procedure A ( procedure F( ... ) )
14        ...
15        F( ... )
16        ...
17    end A
```