UNIVERSITY OF TORONTO
Faculty of Arts and Science

April-May 2007

CSC 488/2107S

Duration – 3 hours (180 minutes)

Examination Aids: Books, Notes, Calculators.

170 marks total, 7 questions on 14 pages. ANSWER ALL QUESTIONS. Write answers in the exam book. If you need to make any assumptions in order to answer a question, state the assumption clearly in your answer book. Good luck!

1. [15 marks] **Grammars and Parsing**.

a. [10 marks] Let L be a language over an alphabet $\{a, b, c\}$ satisfying the following two conditions: (1) a is always immediately followed by at least two c's and (2) b is always followed by at least one c.

i. [4 marks] Write a regular expression for this language.

```
Solution: (c|acc|bc)*
```

ii. [6 marks] Construct a deterministic finite automaton recognizing this language.

```
Solution: the automaton is as follows:

       A  B  C  D

-> A   c  a     b

   B         c

   C   c

   D   c

A is both the initial and the accepting state.
```

b. [5 marks] Write a context-free grammar that accepts the language $L = \{a^n b^m | 2n \geq m \geq n \geq 1\}$.

```
Solution:

S  ->  ab | abb | aSb | aSbb
```

2. [25 marks] **More Grammars and Parsing**. One of the grammars below is LR(0) but not LL(1), one is LL(1) but not SLR(1), one is SLR(1) but neither LL(1) nor LR(0), and one is neither LL(1) nor SLR(1).

I	1:	S	$\longrightarrow$	A S
	2:		$\longrightarrow$	b
	3:	A	$\longrightarrow$	S A
	4:		$\longrightarrow$	a

II	1:	S	$\longrightarrow$	A b
	2:	A	$\longrightarrow$	A a
	3:		$\longrightarrow$	a

III	1:	S	$\longrightarrow$	a S
	2:		$\longrightarrow$	a

IV	1:	S	$\longrightarrow$	A a A b
	2:		$\longrightarrow$	B b B a
	3:	A	$\longrightarrow$	ϵ
	4:	B	$\longrightarrow$	ϵ

a. [15 marks] Explain which is which. Answers without justification are not accepted.

```
Solution:

Grammar I:   neither LL(1) nor LR(0),  because it is an ambiguous grammar.

Grammar II:  LR(0) but not LL(1) - no conflict in LR(0) parse table; it is
             not LL(1) due to left recursion.

Grammar III: SLR(1) but neither LL(1) nor LR(0) - there is a shift/reduce
             conflict during the construction of LR(0) parse table, which
             can be resolved by checking the FOLLOW set of S. It is not
             LL(1) because there are  ``aS'' and ``a'' starting with the
             same terminal.

Grammar IV: LL(1) but not SLR(1) - because FOLLOW(A)=FOLLOW(B)={a,b}, it
            causes a reduce/reduce conflict in the construction of SLR(1)
            parse table. There is no conflict in LL(1) parse table.
```

b. [10 marks] Prove that the grammar that you indicated as LL(1) is in fact LL(1), i.e., compute FIRST and FOLLOW sets and construct the parse table.

```
Solution:

FIRST(S)={a,b}, FIRST(A)=FIRST(B)={\epsilon}
FOLLOW(S)={$}, FOLLOW(A)=FOLLOW(B)={a,b}

         a                b                  $

 S   S->AaAb          S->BbAa

 A   A->\epsilon      A->\epsilon

 B   B->\epsilon      B->\epsilon
```

3. [15 marks.] **Short Questions.** These questions are worth 3 marks each and no mark is given unless a justification is provided.

a. In the constant cost display update algorithm, why is the save/restore operation unnecessary for uplevel calls?

`Because in this case, previous calls (if any) of the functions at the uplevel should be finished. There is no activation record of this level in stack.`

b. True or false: if a grammar has an LR(0) parse table then it has no reduce-reduce conflicts.

`True. If there is reduce/reduce conflict, we cannot determine which production rule should be applied for reduction.`

c. True or false: a single-pass compiler cannot perform dataflow analysis.

`True. Performing dataflow analysis requires visiting codes that have been scanned before, e.g., on programs with loops.`

d. True or false: the mark-and-sweep algorithm is faster than the copying algorithm for garbage collection.

`Either answer is correct as long as an explanation of why one approach is faster than the other is given. For example, mark and sweep is faster if the number of chunks in the heap is smaller than the size of the reachable objects and vice versa for copying.`

e. Given a switch statement where the case variable is an enumerated type with 15 elements and 7 cases appear on the body of the switch, what is the most efficient jump table strategy for translating this statement?

`Hash table with a one-to-one hash function. A branch table wastes more than half the entries. A chain of if statements has a larger code footprint and average run-time cost.`

4. [26 marks] **Data Flow Analysis.** The problem of *use-before-def* refers to the instance where it is possible for a variable to be used before it has been defined. For the following program, assume that

1. declaration is separate from definition, e.g.,

   ```
   int a;
   ```

   does not assign **a** a value, whereas

   ```
   a = 10;
   ```

   clearly does;
2. a variable is only visible within the scope where it is declared; and
3. **E1**, **E2**, and **E3** are boolean expressions that could evaluate to either true or false, and variables present in these expressions are ignored.

Let the following program be given:

```
 1: int x,y,z;
 2: x = 10;
 3: while (E1){
 4:   if (E2) {
 5:     while (E3) {
 6:       int a;
 7:       a = y;
 8:       z = x * 2;
 9:       y = 20+z;
10:    }
11:  }
12:  else {
13:     int a;
14:     y = z+x;
15:     a = x;
16:  }
17: }
18: z = y+x;
```

a. [3 marks] Divide this program into basic blocks and draw the control flow graph.

`B0={1,2} B1={3} B2={4} B3={5} B4={6,7,8,9} B5={13,14,15} B6={18}`

b. [3 marks] Check this program by hand and identify instances of *use-before-def*, if any.

`7: a=y and 18: z=y+x - y may not be defined`

`14: y=z+x - z may not be defined`

c. [3 marks] Is this an all-path or an any-path data-flow analysis problem? Explain your choice.

The problem can posed either as all-path or any-path problem, depending on the analysis you designed.

d. [3 marks] What is the domain of your analysis? Illustrate using the above program.

It can be the set of variables or definitions, depending on the chosen analysis.

e. [6 marks] Design a **backward** data-flow analysis for checking the *use-before-def* problem. Describe what you need to compute and give the data flow equations and initial conditions for **in**[**B**] and **out**[**B**] for each basic block.

We assume each uninitialized variable is assigned a dummy value. Similar to the (backward any-path) live variable analysis, we check if such variable can be used in future along a clear path.

We handle new scope by renaming variables (a at 7 -> a7, a at 13 -> a13).

The equations are as follows:

```
IN[B0]=OUT[B0]-{x,y,z}            OUT[B0]=IN[B1]
IN[B1]=OUT[B1]                    OUT[B1]=IN[B2] \/ IN[B6]
IN[B2]=OUT[B2]                    OUT[B2]=IN[B3] \/ IN[B5]
IN[B3]=OUT[B3]                    OUT[B3]=IN[B1] \/ IN[B4]
IN[B4]=OUT[B4]-{a7,z,y} \/ {y,x}  OUT[B4]=IN[B3]
IN[B5]=OUT[B5]-{a13,y} \/ {z,x}   OUT[B5]=IN[B1]
IN[B6]=OUT[B6]-{z} \/ {y,x}       OUT[B6]={}
```

The initial values of IN[B] and OUT[B] are empty sets.

f. [8 marks] Compute **in**[**B**] and **out**[**B**] for each basic block of the program. Compare the results with those in (b).

IN[0]=OUT[6]={} IN[4]=IN[6]={y,x} IN[5]={z,x}
IN[1,2,3]=OUT[0,1,2,3,4,5]={x,y,z}

y and z are not initialized at B0. But OUT[0]={x,y,z}. Therefore, the y and z with dummy values may be used later. That is, there are use-before-def instances in this program.

Note that such backward any-path analysis only tells us if there ''exists'' use-before-def instances. A forward all-path analysis may give more information, showing the location of the instances.

5. [34 marks] **Handling Dynamic Scoping.** We introduce into the 488 language a dynamic scoping rule. Under this rule, the location of a non-local variable **x** is resolved at run-time to the declaration of **x** in the most recently called routine.

a. [2 marks] What would be the difference in resolving **x**'s location under the static scoping rule?

`Solution:`

`It is resolved to the declaration of x in the innermost enclosing scope.`

b. [4 marks] Let the following program be given:

```
 1: begin
 2:   var x,y : Integer
 3:   proc f of
 4:   begin
 5:     write x, `` ``, y, newline
 6:   end
 7:   proc g (y : Integer) of
 8:   begin
 9:     var x : Integer
10:     x := 3
11:     f
12:   end
13:   x := 1
14:   y := 2
15:   f
16:   g (3)
17: end
```

i. [2 marks] What is the output of this program under static scoping?

ii. [2 marks] What is the output of this program under dynamic scoping?

```
i) under static scoping
Solution:
1 2
1 2
ii) under dynamic scoping
Solution:
3 3
3 3
```

c. [8 marks] List four compile-time changes required to support dynamic scoping. In particular, describe the changes to semantic analysis and the activation record layout.

`A possible solution:`

```
-- visibility check for use of x in routine r becomes: ``check that x is
declared in all calling contexts of r (sequences of calls terminating in r)''
-- type check for use of x in routine r becomes: ``check that the type of x in
all calling contexts of r is compatible with the use''
```

```
-- these first two changes could be combined as: ``compute all calling
contexts of r, change symbol table lookup for use of x in r to traverse all
calling contexts of r; if x is not visible in some context: error; otherwise,
if all declarations of x are compatible with use: ok''
-- this approach statically ensures type correctness; an alternative is to do
type-checking at runtime and raise an exception in case of a type error

-- add to activation record a dynamic link (pointer to base of caller's
activation record)
-- add to activation record either:
  -- an array mapping variables/ids to offsets or UNDEF; this requires
  specifying a variable order and wastes space in the AR for variables not
  declared in the correponding routine
  -- a hash table providing the same map; this requires computing hashes at
  compile-time and resolving collisions but decreases or eliminates wasted
  space in the AR
```

d. [10 marks] Describe an algorithm for computing the address of a variable **x** at run-time.

```
Solution:

let id_x be the id of x
let P be the address of the base of the current AR
offset_x := UNDEF
while (true) {
  offset_x := P->lookup_table[id_x]
    % or offset_x := lookup_subroutine (P->lookup_table, id_x)
  if (offset_x != UNDEF)
    break
  P := P->dynamic_link
}
return &P->control_block + sizeof(P->control_block) + offset_x
```

e. [10 marks] Using your solutions to parts (d) and (e), show the run-time stack configuration for the example above after procedure **g** has called procedure **f**, and line 5 is about to be executed. Briefly describe how the address for **x** is resolved.

```
Solution:

Using an array to map variable ids to offsets:

f::x UNDEF
f::y UNDEF
f::offset_x UNDEF
f::offset_y UNDEF
link to g's control block
base of f's control block
```

```
g::x 3
g::y 3
g::offset_x 3
g::offset_y 2
link to main's control block
base of g's control block
...
main's AR

Lookup offset_x in f's lookup table, get UNDEF
Follow the link to g's control block
Lookup offset_x in g's lookup table, get 3
Return (address of top of g's control block + 3)
```

6. [25 marks] **Code Generation Templates.**

a. [10 marks] We introduce into the 488 language the construct **foreach** to express iteration over arrays. An example of the use of this feature is shown below:

```
var A[0..4] : Integer
var i : Integer
i := 0
foreach x in A do
begin
  x := i
  i := i + 1
end
foreach x in A do
begin
  write x, `` ''
end
```

This program outputs "0 1 2 3 4". Notice that the iterator variable, **x**, is used both to read from and write to the array **A**.

Specify code templates for (1) a **foreach** loop instance and (2) a use and (3) a definition of a **foreach** iterator variable.

```
Solution:

Where an array A is declared as
var A[L..U] : T

rewrite an instance of a foreach loop

foreach x in A do
begin
  %% body %%
end
```

```
as

var I : Integer % fresh declaration in enclosing scope
I := L
while I < U + 1 do
begin
  %% body %%
  I := I + 1
end

and replace all occurences of x in %% body %% with A[I]. Emit code as before.
```

b. [15 marks] We introduce into the 488 language pointer types (e.g., *Integer) and dereference (**@**) and address-of (**&**) operators. A variable of pointer type ***T** stores the address of a variable of type **T**. The dereference operator, applied to a pointer variable, obtains the value at the address stored in the variable. The address-of operator returns the address of a variable. For example, the program

```
begin
  var x, y : Integer
  var ip : *Integer

  x := 5
  ip := &x
  y := @ip
  write y, `` ''
  @ip := 3
  write x
end
```

outputs "5 3".

Specify code generation templates for:

i. [5 marks] the address-of operator.

ii. [5 marks] the dereference of a variable appearing on the left-hand side of an assignment, and

iii. [5 marks] on the right-hand side of an assignment.

```
Solution:

Below, assume x has lexical coordinates <l,d>, ip has coordinates <k,r>, and
the size of the control block is a constant C:

(i) &x

As before, emit instructions to compute the address of x using the display and
arithmetic.
```

```
PUSH C
ADDR l d
ADD
```

Now the address of x is on the top of the RTS. This differs from the code template for evaluating a scalar in that it omits the final LOAD instruction.

(ii) @ip on LHS

```
PUSH C
ADDR k r
ADD      %% top = &ip
LOAD     %% top = ip
```

This differs from the template for a scalar on the LHS in that it includes a final LOAD instruction.

(iii) @ip on RHS

```
PUSH C
ADDR k r
ADD       %% top = &ip
LOAD      %% top = ip
LOAD      %% top = @ip
```

This differs from the template for a scalar on the RHS in that it includes an additional final LOAD instruction.

7. [30 marks] **Runtime environment: structs and unions.**

a. [10 marks] Apply the "depth first" structure mapping algorithm to the following struct declaration:

```
struct {
  short Z;
  union {
    double T;
    int R;
    char X[6];
  } U;
  char* S;
  struct {
    char c;
    short Y[2];
  } first [2];
} mystruct;
```

Assume the datatypes have the following sizes and alignment constraints:

```
type        size      align
char*       32        32
double      64        64
int         32        32
char        8         8
char[K]     8*K       8
short       16        16
short[K]    16*K      16
```

```
Solution:

name     item   align     length    align_i   length_i          fill_i
U:
T        2,1                        64        64                0
R        2,2                        32        32                0
X        2,3                        8         6*8=48            0
                 64        64

first:
c        3,1                        8         8                 0
                 16        8
Y        3,2                        16        2*16=32           8
                 16        48

mystruct:
Z        1,1                        16        16                0
                 64        16
U        1,2                        64        64                48
                 ``        128
S        1,3                        32        32                0
                 64        160
first    1,4                        16        2*48=96           0
                 ``        256

Each character is 8 bits, f is ``first'', _ is fill:

0 1     6       1   1 1   2 2   2
  6     4       2   6 7   0 2   5
                8   0 6   8 4   6

ZZ______UUUUUUUUSSSSffffffffffff
  TTTTTTTT          c_YYYYc_YYYY
  RRRR
  XXXXXX
```

b. [5 marks] Re-order the fields in order to minimize fill.

```
Solution:

struct {
  union {
    double T;
    int R;
    char X[6];
  } U;
  char* S;
  struct {
    char c;
    short Y[2];
  } first [2];
  short Z;
} mystruct;

0       6   9           1 2
        4   6           9 0
                        2 8

UUUUUUUUSSSSfffffffffffffZZ

Note that the fill in first is not eliminated by reordering because first is
an array of structs; fill would still be required between the elements of
first if c and Y were swapped.
```

c. [15 marks] Using the addresses computed by your structure mapping in part b., outline a series of optimizations that can be performed on the following code:

```
...
struct mystruct s;
...
for (i = 0; i < 6; i++) {
  for (j = 0; j < 2; j++) {
    s.U.X[i] = s.first[j].c;
    s.first[j].Y[j] = s.first[j].Y[j] + s.Z;
  }
}
```

You do not have to show how the code looks after every individual optimization, although you have to describe what each applicable optimization accomplishes and give the final code. Assume that all the fields of **s** are initialized and that addition is cheaper (faster) than multiplication. Also, unroll the innermost loop twice (thereby eliminating that loop altogether). Note that the address unit here is the *byte*, whereas the units in the previous question are *bits*. Assume there are eight bits in a byte.

```
Solution:

-- declare a base address for s and use the solution from b) to compute
```

```
offsets to the fields and subscripts

addrS = &s;
addrSZ = addrS // s.Z
addrSU = addrS + 2
addrSUV = addrSU
addrSUVi = addrSU + i = addrS + 2 + i // s.U.V[i]
addrSS = addrS + 16
addrSf = addrS + 20
addrSfi = addrSf + 6*i
addrSfic = addrSfi = addrSf + 6*i = addrS + 20 + 6*i // s.first[i].c
addrSfiY = addrSfi + 2
addrSfiYj = addrSfiY + 2*j
  = addrSfi + 2 + 2*j = addrS + 20 + 6*i + 2 + 2*j // s.first[i].Y[j]

-- apply all relevant optimizations in the optimization tutorial handout

-- it should look something like this afterwards:

original:

for (i = 0; i < 6; i++) {
  for (j = 0; j < 2; j++) {
    s.U.V[i] = s.first[j].c;
    s.first[j].Y[j] = s.first[j].Y[j] + s.Z;
  }
}

expand subscripts:

for (i = 0; i < 6; i++) {
  for (j = 0; j < 2; j++) {
    *(addrS + 2 + i) = *(addrS + 20 + 6*j);
    *(addrS + 20 + 6*j + 2 + 2*j) = *(addrS + 20 + 6*j + 2 + 2*j) + *addrS;
  }
}

factor out CSEs:

addrS2 = addrS + 2;
addrS20 = addrS + 20;
addrS22 = addrS + 22;
for (i = 0; i < 6; i++) {
  for (j = 0; j < 2; j++) {
    addrS22_8j = addrS22 + 8*j;
```

```
    *(addrS2 + i) = *(addrS20 + 6*j);
    *addrS22_8j = *addrS22_8j + *addrS;
  }
}
```

unroll inner loop:

```
addrS2 = addrS + 2;
addrS20 = addrS + 20;
addrS22 = addrS + 22;
for (i = 0; i < 6; i++) {
  addrS22_8j = addrS22;
  *(addrS2 + i) = *(addrS20);
  *addrS22_8j = *addrS22_8j + *addrS;
  addrS22_8j += 8;
  *(addrS2 + i) = *(addrS20 + 6);
  *addrS22_8j = *addrS22_8j + *addrS;
}
```

code motion, useless code elimination, and variable propagation:

```
addrS2 = addrS + 2;
addrS22_8j = addrS + 22;
x = *addrS;
*addrS22_8j = *addrS22_8j + x;
addrS22_8j += 8;
*addrS22_8j = *addrS22_8j + x;
v = *(addrS20 + 6);
for (i = 0; i < 6; i++) {
  *(addrS2 + i) = v;
}
```

END OF EXAM