University of Toronto

CSC 488/2107 Language Processors **Winter 2005**
Midterm Test [15% of final mark] **Feb. 24, 2005**

Total marks 90 - Total time is 50 minutes. Answer all 6 questions.
Instructions: This midterm is open book, open notes. Non-programmable calculators allowed. No electronic communication devices allowed.
The line numbers in grammars are for reference only and are not part of the grammars. Ellipses ... indicate omitted, correct text. ϵ indicates an empty string.
If you need to make any assumptions in order to answer a question, state the assumptions clearly in your answer book.

1 [15 marks + bonus]. Languages and automata.

a. [**12 marks**] Create an automaton recognizing a language over an alphabet $\{0, 1\}$ where the number of 0's is odd AND the number of 1's is odd.

```
The trick was to realize that you will need four states, for keeping
track of ''evenness'' and ''oddness'' of 0, and 1.  We refer to
these states as :
   A - (e, e)    /* both are even */
   B - (e, o)    /* 0 even, 1 odd */
   C - (o, o)    /* both odd      */
   D - (o, e)    /* 1 even, 0 odd */

Then the automaton is as follows:

    A   B   C   D
  A     1       0
  B 1       0             A is the initial state.
  C     0       1         C is the accepting state
  D 0       1

Marking:
  12 marks for a correct answer
   8 for an automaton that is close to the correct one
```

b. [**3 marks**] Is this language regular? I.e., can it e described by a regular expression?

```
Solution
The language is regular since it is recognized by an automaton. For every
regular language, there is a corresponding regular expression.

Marking:
   3 marks for saying 'Yes'
```

c. [**Bonus: 6 marks.**] If you answered yes to the previous question, give the regular expression for this language.

Solution. Here is the best solution we have seen:

The idea is to recognize that expressions we are after are the
ones that can be divided into
  <even # of 0s and 1s> 1 <even # of 0s and 1s> 0 <even # of 0s and 1s> |
  <even # of 0s and 1s> 0 <even # of 0s and 1s> 1 <even # of 0s and 1s>

And the expression for <even # of 0s and 1s> is:
  (00 | 11 | 0101 | 1001 | 1010 | 0110)*

Marking:
  4 marks for a reasonable but incorrect answer

2 [15 marks]. Construct a DFA for a language

1: S $\longrightarrow$ B C D D
2: B $\longrightarrow$ D B | ϵ
3: C $\longrightarrow$ '+'
4: D $\longrightarrow$ '+' | '−'

There are several parts to this question. First is to recognize that
this grammar specifies a regular expression:
  (+ | -)* + (+ | -) (+ | -)

and thus the NFA is obvious. It has 4 states (A, B, C, D)

```
   A    B    C    D
A +/-  +                    A is the initial, D is accepting
B          +/-
C               +/-
D
```

The goal then is to convert this NFA into a DFA using subset construction.

The answer has states A - {1}, B - {1,2}, C - {1,3}, D - {1,4},
  E - {1,2,3}, F - {1,2,4}, G - {1,3,4}, H - {1,2,3,4}

A is initial, D, F, H are final

```
   A  B  C  D  E  F  G  H
A -  +
B       -     +
C          -     +
D -  +
E                    -  +
F       -     +
G          -     +
```

```
  H                  -  +

Marking:
8 marks for NFA
   4 marks if just the regular expression
7 marks correct construction of DFA using subset construction
   4 marks for an attempt
```

3 [20 marks]. Show that the following grammar is LR(1) but not LALR(1):

1: S $\longrightarrow$ p X q
2: $\longrightarrow$ X r
3: $\longrightarrow$ p Y r
4: $\longrightarrow$ Y q
5: X $\longrightarrow$ w
6: Y $\longrightarrow$ w

```
Marking:
  10 marks for LR(1)
      10 marks for a CFSM or a table (12 states)
      or
      6 marks for a convincing argument but no table
      or
       8 marks for an LR(0)-looking table

  10 marks for not LALR(1)
     You have to show that there is a reduce/reduce conflict
     on rules 5 and 6 that have an identical lookahead set.
```

4 [15 marks]. What errors may occur in the following piece of code? (Here we are using a C-like language but with nested method scopes.) Describe which semantic checks will catch these errors. Separate your list into static and run-time checks.

```
      main() {
         ...
         void mine () {
           ...
           int A = B[i+3] * C - compute(B,C);
```

```
There are many possible problems, including:
  - B/C/i not visible
  - mine/A might have be defined previously
  - compute not defined
  - compute defined, but not as a function
  - compute takes a different number of parameters
  - types of B,C might not match compute's argument types
```

```
  - B is not an array
  - i and 3 can't be added
  - value of B[i+3] not compatible with C (*)
  - value of (B[i+3]*C) not compatible with compute(B,C) (-)
  - lvalue, rvalue on assignment aren't compatible
  - i+3 is not within bounds of B
  - overflow during arithmetic operations

The last two problems need to be checked at run-time; the others are
checked at compile-time. Answers with just checks or just errors were
acceptable.

Marking:
  1.5 for each correct check or error
  -0.5 for each incorrect check or error (max 5)
  -0.5 for an incorrect identification of run-time / compile-time (max 5)
```

5 [10 marks]. Let the following Pascal type declaration be given

```
type  link = ↑ cell;
      cell = record
           info : integer;
           next : link
      end;
```

Here, link is a pointer to cell, and its type is *pointer*(cell). Type of a tuple A : T is indicated as A × T, and type of a record with fields B and C is indicated as *record*(B × C). Listed below are seven type expressions. Which of these are name equivalent and which are structure equivalent? For your answer, you can give a 7 × 7 table, indicating each entry as S (structure), N(name) or blank (neither).

1. link
2. *pointer*(link)
3. cell
4. *pointer*(cell)
5. *record* ((info × integer) × (next × *pointer*(cell))
6. *pointer*(*record* ((info × integer) × (next × *pointer*(cell))))
7. *pointer*(*record* ((info × integer) × (next × link)))

```
Solution:
    1  2  3  4  5  6  7
------------------------
  1 N        S     S  S
```

```
  2    N
  3       N     S
  4 S        N     S  S
  5       S     N
  6 S        S     N  S
  7 S        S     S  N
```

Certain things can be subject for interpretation, i.e., record and cell have been explicitely defined to be the same. Are they N or S? Both got full marks.
You got 3 marks for N on the diagonal, +1 for a right entry, -1/2 for the wrong.

6 [15 marks]. Suppose we are working with an n-dimensional array A[0..u_1, 0..u_2, ..., 0..u_n]. A is *sparse*: non-zero entries are allowed only on the diagonal. Show how efficient array storage for such sparse arrays can be laid out by the compiler. How much space would be necessary? What is the general case of an array subscript calculation, i.e., how a reference to an array element A[E_1, E_2, ..., E_n] is transformed into a memory address of this array element?

Most of you noted that we only need n entries to store (n+1, with the additional entry storing 0, was acceptable as well).

The address is computed as follows:

```
A(s1, s1, ..., s1) = A'[s1] if s1 <= min (u1, ... un)
                     0 otherwise
General array subscript calculation for A'[s1]
under the above conditions is
  base + size of each element * s1
```

Marking:
- 2 marks for noting this takes n elements
- 10 marks for showing how this is done:
  - 5 marks for general idea
  - 3 marks for indicating that otherwise the value is 0
  - 0 marks for noting that this works for s1 <= min (u1, ..., un)
- 3 marks for array subscript calculation. The point here was not to forget the size of each element