RYERSON POLYTECHNIC UNIVERSITY

DEPARTMENT OF MATH, PHYSICS, AND COMPUTER SCIENCE

CPS 710

FINAL EXAM

FALL 97

STUDENT ID: _____

INSTRUCTIONS

Please write your student ID on this page. Do not write it or your name on any other pages.

Please answer directly on this exam.

This exam has 4 questions, and is worth 30% of the course mark. It has 8 pages including this one

NO AIDS ARE ALLOWED.

A - General Concepts	30
B - Shift-Reduce parsing	20
C - Evaluation	25
D - Grammars	25

Part A - General Concepts - 30 marks

A1 (7 marks)

Draw a transition diagram to recognise the regular expression **a (a | b)* b* a**

Show all **a** and **b** edges from all states of your diagram

A2 (4 marks)

Give two reasons why you would want to have the scanning module separate from the rest of the program.

A3 (4 marks)

What is a parser?

A4 (5 marks)

List 5 fields kept in a symbol table entry

A5 (4 marks)

List 4 errors that can only be detected once the symbol table is implemented.

A7 (6 marks)

What would be the output of the following program fragment:

```
integer a = 1;
integer b = 2;
function f( integer a)
    {
    b := a + 5;
    return b;
    }
function g(integer a)
    {
    integer b;
    return f(a);
    }
display g(6);
display b;
```

If the language is **statically scoped**?	If the language is **dynamically scoped**?

Part B - Shift-Reduce Parsing (20 marks)

Given the LR(1) grammar with the following productions augmented with states (shown subscripted and outlined):

(1) CASE → $_{1}$ **case** $_{2}$ **id** $_{3}$ CLAUSES $_{8}$ **end** $_{10}$

(2) CLAUSES → $_{3}$ CLAUSES $_{8}$ CLAUSE $_{9}$

(3) CLAUSES → $_{3}$ CLAUSE $_{7}$

(4) CLAUSE → $_{3,8}$ **constant** $_{4}$ **:** $_{5}$ **stat** $_{6}$

Fill in the LR table for this grammar:

	CASE	CLAUSES	CLAUSE	**case**	**id**	**end**	**constant**	**:**	**stat**	**$**
1										
2										
3										
4										
5										
6										
7										
8										
9										
10										

There are 4 possible entries for each slot in the table:

- S state: shift token onto symbol stack and change state
- R production: reduce the production by popping n symbols off the symbol stack and n symbols off the state stack, and inserting the left-hand side of the production into the input stream
- HALT: to halt process
- nothing: to indicate a syntax error

Part C - Evaluation - 25 marks - This has a short answer

In this question you will be evaluating simplified DO expressions in Scheme.

- The section of the grammar which deals with DO expressions is:

DO -> **(do (** INDEX-LOOP**)**
(EXIT-CONDITION FINAL-VALUE **)**
DO_BODY **)**
INDEX-LOOP -> **(** VAR-NAME INITIAL-VALUE NEXT-VALUE **)**

VAR-NAME -> **Identifier**
INITIAL-VALUE -> EXPRESSION
NEXT-VALUE -> EXPRESSION
EXIT-CONDITION -> EXPRESSION
FINAL-VALUE -> EXPRESSION
DO-BODY -> EXPRESSION*

Tokens are in **bold-face**. The non-terminal EXPRESSION is defined somewhere else in the grammar. No other productions use the INDEX-LOOP non-terminal.

- A syntax tree node produced by the parser has the structure:

```
typedef struct node {
    int type;
    struct node *v1, *v2, *v3 *v4;
    } node;
```

- a DO structure is organised as follows:
  - type = DO
  - v1 points to an INDEX_LOOP structure
  - v2 points to the exit condition which must evaluate to TRUE for the loop to exit
  - v3 points to the final value of the entire loop
  - v4 points to the do body which is evaluated at each loop iteration

- an INDEX_LOOP structure is organised as follows:
  - type = INDEX_LOOP
  - v1 points to the name of the loop index
  - v2 points to the initial value of the loop index
  - v3 points to an expression describing how to calculate the next value of that index at each iteration
  - v4 is NULL

- The evaluation rules for these two structures are:
  1. The loop index is initialised, which means that a new index variable is created whose name is defined in the INDEX_LOOP v1 field and initial value in its v2 field. The symbol table functions used to do this are: (you can assume identifier tokens are of type *node)
     void newsymbol(node *identifier), and void setvalue(node *identifier, int newvalue)
  2. The EXIT-CONDITION is evaluated
     - 2.1 If it evaluates to TRUE, the final value is evaluated and returned
     - 2.2 Otherwise
       - 2.2.1 The do body is evaluated
       - 2.2.2 The next value is evaluated and the loop index set to it
       - 2.2.3 Continue execution at 2.

- You are writing int eval(node *tree), a function which evaluates a syntax tree. Eval returns an int which represents the value of the syntax tree (you can assume that all values of all types can be represented by integers). Eval is completely written except for the part which handles DO and INDEX_LOOP structures.

- Write C code for the section of eval which handles DO structures, and whatever C code is necessary to handle INDEX_LOOP structures.

- You do not need to do any memory management in your code. You can assume that the DO expression you are evaluating is error-free.

Part D - Grammars - 25 marks

In this question, non-terminals are in UPPER-CASE and terminals are **underlined.**

D1 (6 marks)

What is an **ambiguous** grammar? Why would you not want a grammar to be ambiguous?

D2 (6 marks)

Why is the following set of productions **ambiguous**?

CASE → **case identifier in** CLAUSES **endcase**

CLAUSES → CLAUSE CLAUSES | ε

CLAUSE → EXPRESSION **:** STATEMENT

STATEMENT → **identifier :=** EXPRESSION

EXPRESSION → (+ | −)^{0} **constant** {(+ | −) EXPRESSION}^{0}

D3 (6 marks)

Left-factor the following set of productions fully. You may need to introduce new non-terminals.

B → C | **t** | **f**

C → (**t** | **f**) $\veebar$ C

C → (**t** | **f**) $\barwedge$ C

D4 (6 marks)

Remove the **right recursion** from the following set of productions. You may need to introduce new non-terminals. You can make the grammar left recursive if you wish.

S → E { $\pm$ S }^{0}

E → T { ***** E }^{0}

E → **(** S **)**

T → **0** | **1** | **2**