Name: _____

Preliminary Examination 1
CS412: Introduction to Compilers

5 March 2002

Instructor: Radu Rugina
Cornell University, Computer Science Department

The duration of the exam is 2 hours.
There are 5 parts, each counting 20 points. In the first part you have to define and compare several basic concepts. The following 4 parts consist of problems that cover the main topics discussed in class.
You should have 9 pages for this exam.

---

1. [20pts] Basic Concepts

a) [4 pts] What is the difference between a deterministic finite automata (DFA) and a non-deterministic finite automata (NFA)?

b) [4 pts] What is an ambiguous grammar?

c) [4 pts] What is an LR(1) state and what does it represent?

d) [4 pts] What is a strongly typed language?

e) [4 pts] What is a syntax-directed definition?

2. [20 pts] Regular Expressions and Finite Automata

Consider the language of strings a's and b's, which have even length and where the characters in the odd positions in the string are always different than their successor characters (the index of the first character in the string is 1).

a) [5 pts] Write a regular expression that accepts this language.

b) [5 pts] Construct the NFA for this regular expression. Use ε-transitions to build the NFA for Kleene star and alternation expressions.

c) [5 pts] Convert the constructed NFA to a DFA. Show the correspondence between the NFA states and the DFA states.

d) [5 pts] Minimize the number of states in the constructed DFA. Show the correspondence between the states in the original DFA and the states in the minimized DFA.

3. [20 pts] Grammars and Parsing

Consider the following grammar :

```
S → E $
E → E(E)  | E[E] |  id
```

a) [5 pts] Show that this is not an LL(1) grammar and build an LL(1) grammar which accepts the same language. Explain how you constructed your LL(1) grammar.

b) [6 pts] Determine the nullable non-terminals and construct the FIRST sets. Next write the inclusion constraints for FOLLOW sets. For each constraint, show the production which generated the constraint. Then use these constraints to compute the FOLLOW sets.

c) [4 pts] Build the LL(1) parsing table for the grammar you derived in part b).

d) [5 pts] Consider now that you want to build an LR(0) parser for the original grammar. This requires to build the parser DFA consisting of LR(0) states and transitions between these states. Show the DFA states that the parser traverses when it builds the expression E [E] on the stack.

4. [20 pts] Type Checking

Consider a language which manipulates integer expressions, Boolean expressions, and stack expressions. The grammar below defines the types and the expressions in this language:

```
T → int | bool | stack(T)
E → id | num | E + E
E → top(E) | pop(E) | push(E,E) | newstack(T) | empty(E)
```

The type `stack(T)` defines a stack of elements of type `T`. The addition operation `E+E` can be applied to integers or stacks. You can add two integer numbers to produce an integer value which is the sum of the two numbers. You can also add two stacks `E` and `E'` if they have elements of the same type. The result will be a stack `E"`, which has all the elements of `E'` on top of `E`. For a stack `E`, `top(E)` is the element on top of the stack, and `pop(E)` is the stack resulted after removing the top element from `E`. For a stack `E` with elements of type `T`, and for an element `E'` of type `T`, `push(E,E')` adds the element `E'` on top of the stack `E` and returns the resulting stack. The constructor `newstack(T)` produces a new, empty stack of type `T`. Finally, for a stack expression `E`, `empty(E)` is true if `E` has no elements, and false otherwise.

a) [8 pts] Write the type-checking rules for all the expressions in this language.

b) [6 pts] Which of the following expressions are well-typed, and in what context? Indicate the result type for each well-typed expression.

i) `push(x,empty(y)) + z`

ii) `push(newstack(int), top(x)) + push(y,empty(z))`

iii) `push(top(x), pop(y)+newstack(int))`

c) [6 pts] Write the typing proof tree for one of the well-typed expressions in part b).

5. [20 pts] IR Generation

Consider a language with for loops and repeat statements:

`S` → `for(v=E1 to E2 step E3) S | repeat S until E`

The "`for`" loop construct first initializes the integer index variable `v` with the value of integer expression `E1`. It then iteratively executes statement `S`. At each iteration, it first executes `S` and then increments the value of `v` by `E3`. The execution of the loop terminates when `v` exceeds the value of the upper bound expression `E2`.

The "`repeat`" construct first executes `S`, then tests the Boolean condition expression `E`. If the condition evaluates to true, the execution of the loop terminates. Otherwise, it proceeds to the next iteration.

a) [7 pts] Assume the high-level intermediate representation does not have nodes for repeat constructs and for loops, but only while loops: `while (E) S`. The compiler must therefore translate repeat and for loops into while loops. Write a syntax-directed definition which describes the translation of for loops and repeat constructs into high-level IR with while loops only.

   For the translation, assume the compiler has the following high-level IR statement nodes:

   `Block(S, ..., S), While(E, S)`

   and the following high-level IR expression nodes:

   `Assign(v, E), LessThan(E, E),`
   `Add(E, E), Negate(E), VarExpr(v)`

   In the above IR nodes, `E` denotes an expression, `S` denotes a statement, and `v` denotes a variable.

b) [7 pts] Write the translation of the for loop construct to low-level intermediate code. Use the notation ⟦ `S` ⟧ to express the low-level code for statement `S` and `t` = ⟦ `E` ⟧ for the low-level code which computes the value of integer or Boolean expression `E` and stores the result in `t`.

c) [6 pts] Write a translation of the repeat construct, which encodes the condition `E` into the control-flow. Use the notation ⟦ `E, L, L'` ⟧ to denote the low-level intermediate code for Boolean expression `E`, which jumps to `L` if `E` evaluates to true and jumps to `L'` if `E` evaluates to false.