CPSC 411, Fall 2010 – Midterm Examination

Name: _____

Q1:	10
Q2:	15
Q3:	15
Q4:	10
Q5:	10
	60

Please do not open this exam until you are told to do so. But please do read this entire first page now.

Some of the problems on this quiz ask for written answers, rather than code or numbers. The <u>best answers to such questions are short, concrete and to-the-point</u>. Sometimes a small example program can help to make the answer clear.

Do not attempt to write 'cover all the bases' answers to such questions – overly long answers will receive no marks.

Write all your answers on the <u>front side</u> of each page.

1. High Level Question [10 pts]

So far we have covered 4 compiler stages. The initial input is a program in textual form. The final output is IR. In the space below, in order from first to last, write the name of each stage and briefly describe what it does.

Stage 1 name: ___Lexical Analysis___

input: stream of characters   output: __Stream of tokens___

description:

Divides the input into tokens. Tokens are "word-like" entities in the input, such as identifiers, numbers, operators, punctuation marks etc. It is also the job of this stage to skip over whitespace and comments.

Stage 2 name: __Parsing__

input: ___Stream of Tokens___   output: ___Parse tree (AST)__

description: This stage analyzes the "phrase structure" of the program. It constructs a parse tree or abstract syntax tree. that represents this structure, while abstracting away from the details of the concrete syntax.

Stage 3 name: ____Semantic Analysis_____

input: ___Parse Tree___   output: ____Decorated AST + error messages + symbol table___

description: This stage's main function is to check for "semantic errors", such as type errors and undefined identifiers. It may also decorate the AST with extra information (e.g. expressions may be decorated with their type).

The stage makes heavy use of tables (symbol tables) that keep track of information about identifiers defined in a particular context.

Stage 4 name: ____Translation to IR_____

input: __Decorated AST, symbol table_____   output: IR Trees

description: This converts a (assumed to be semantically correct) program into equivalent IR code. IR code is an "intermediate representation" that is at the same time closer to assembly language and also abstract enough to not be specific to a particular target architecture or source language.

2. Symbol Tables [4pt]

Below are two simple Java interfaces for symbol tables. The differences are highlighted in **bold**.

```
interface XXXTable<Value> {
   Value lookup(Symbol key);
   void insert(Symbol key, Value v);
}

interface YYYTable<Value> {
   Value lookup(Symbol key);
   YYYTable<Value> insert(Symbol key, Value v);
}
```

a) [1pt]Which of the the two is the **functional** style: XXX / YYY (circle one)

b) [1pt]One of the two interfaces is incomplete: it requires a way to remove symbols from the table. To provide this functionality we might provide two additional methods:

```
void beginScope();
void endScope();
```

Which table needs these two methods? Functional / Imperative (circle one)

c) [2pt] Explain briefly **why the other style table does not** require any methods for removing entries. (Note: there is much more space below here than you really need).

The reason we need a "delete" is to be able to "undo" inserts when we are leaving a scope and need to go back to a higher scope. In the functional style we don't need an undo to restore the symbol table to a previous scope, because in the functional style the table from the previous scope still exists (an insert creates a new table without modifying the original table, so the original table still exists).

Symbol Tables continued [11pts]

We can use either hashtables or trees to implement each style of symbol tables but of course one is more natural than the other and yields better performance.

Below provide **"big O" estimates** of the expected performance of a good quality implementation.

<u>Important note:</u> hash tables are implemented using side effects. A correct implementation of a "functional style interface" may use side-effects internally, but from a user's point of view it must behave as a purely functional implementation.

a) [4pt] Estimate the **time consumed** by each operation:

	Functional style interface		**Imperative** style interface	
	lookup	insert	lookup	insert
Tree-based	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Hash-based	O(1)	O(n)	O(1)	O(1)

b) [2pt] Estimate the **memory consumed by the insert** operation..

	Functional style interface	**Imperative** style interface
Tree-based	O(log(n))	O(1)
Hash-based	O(n)	O(1)

Note: "memory consumed" means how much additional memory, on average, would be allocated during the execution of the insert operation. Do not consider potential garbage collection that may (or may not) occur after the insert operation is complete.

c) [5pt] Assume that we are implementing an efficient single pass compiler for Pascal. All you need to know about Pascal for this question is that Pascal was designed to make it easy to implement a **single pass compiler** (the declaration of any identifier must always occur before it is first used).

What style of interface would you choose.   Functional / Imperative   (circle one)

What implementation do you choose.   Hash   / Tree   (circle one)

Explain briefly why:

Generally the imperative hash-based implementation is the most efficient (in terms of both time as well as space). The only reason to choose a functional/tree style interface is because it supports persistence in an efficient manner (at the cost of making inserts and lookups a bit more expensive).

Since our compiler is "single pass" it will only enter/exit each scope in the program exactly once. We would have little use for the "persistence" feature. So we have no good reason to choose the less efficient functional / tree-based implementation.

3. Parsing [4pts]

The following is an EBNF definition for a simple expression language.

```
Exp   ::= Exp ("+"|"-") Exp
        | Exp ":" Exp
        | "(" Exp ")"
        | <IDE>
        | <NUM>
<IDE> ::= <LETTER> ( <LETTER> | <DIGIT> )*
<NUM> ::= (<DIGIT>)+
```

This syntax is inspired by the Haskell language. The ":" operator is an infix version of Scheme's cons operator for creating lists. For example, assuming the identifier `empty` is bound to a representation of the empty list, a list of three elements [1,2,3] can be constructed as follows:

```
1:2:3:empty
```

a) This grammar is ambiguous. Give an example that demonstrates this and explain the example by drawing parse trees.

There are many examples... essentially, any expression that uses at least two operators will do.

A simple example is

1+2-3

It can be parsed as

```
   +            or            -
  / \                        / \
 1   -                      +   3
    / \                    / \
   2   3                  1   2
```

Thus, the grammar is ambiguous since there are some Strings in the language that can be parsed in more than one way.

Parsing (continued) [6pts]

To make the grammar unambiguous we should take operator precedence and associativity into account. The table below shows the intended precedence rules.

	Example	Equivalent to
Rule 1: "+" and "-" associate left	`3+4-5+6`	`((3+4)-5)+6`
Rule 2: ":" associates right	`1:2:3:empty`	`1:(2:(3:empty))`
Rule 3: "+" and "-" highest precedence	`1+2:3-4:empty`	`(1+2):(3+4):empty`

Below is an excerpt from a JavaCC definition.

```
// BNF in left column                  // Java code in right column

Exp Exp() :                            { Token t; Exp e1,e2;}
{ e1=SimplerExp()
  ( (t="+"|t="-") e2=SimplerExp()      { e1 = new OpExp(t.image, e1, e2); }
  )*                                   { return e1;}
}

Exp SimplerExp() :                     { Token t; Exp e1,e2;}
{ e1=PrimExp()
  ( t=":" e2=PrimExp()                 { e1 = new OpExp(t.image, e1, e2); }
  )*                                   { return e1;}
}

Exp PrimExp() :                        { Exp e; }
{
(     e=Num()
|     e=Ide()
|     "(" e=Exp() ")"
)                                      { return e;}
}
```

b) The JavaCC file compiles without error but it has logical bugs. Below, answer for each precedence rule whether the parser correctly follows that rule.

Rule 1: "+" and "-" associate left   Correct / Incorrect   (circle one)

Rule 2: ":" associates right   Correct / Incorrect   (circle one)

Rule 3: "+" and "-" highest precedence   Correct / Incorrect   (circle one)

Parsing (continued) [5pts]

On this page, write a correct replacement for the JavaCC code on the previous page.

Two things need to be done:

      - switch the "+"|"-" tokens with the ":" to fix precedence between them

      - then fix the ":" rule to make it build the parse tree right to left

```
Exp Exp() :                             {Token t; Exp e1,e2;}
{ e1=SimplerExp()
  ( t=":" e2=SimplerExp()               { e1 = new OpExp(t.image, e1, e2); }
  )?                                    { return e1;}
}

Exp SimplerExp() :                      {Token t; Exp e1,e2;}
{ e1=PrimExp()
  ( (t="+"|t="-") e2=PrimExp()          { e1 = new OpExp(t.image, e1, e2); }
  )*                                    { return e1;}
}

Exp PrimExp() :
... unchanged...
```

4. Translation into IR Code [10pts]

Using the IR syntax provided in the appendix, write out plausible IR code that a hypothetical translator might produce. You may assume that the code is from a valid MiniJava program and that any variables in the code are allocated to a TEMP of the same name (e.g. x -> TEMP(x))

a) [2pts]
```
sum = sum+i;
```

```
MOVE( TEMP(sum), PLUS( TEMP(sum), TEMP(i) ) )
```

b) [2pts]
```
i = i + 1;
```

```
MOVE( TEMP(sum), PLUS( TEMP(sum), CONST(1) ) )
```

c) [2pts] (Hint: you may "reuse" the code from a and b here by drawing a box labeled with a or b.)
```
while (i<10) { sum = sum+i; i=i+1; }
```

```
SEQ(  LABEL(start),
      CJUMP(LT, TEMP(i), CONST(10), body, end),
      LABEL(body),
      [A],
      [B],
      JUMP(NAME(start)),
      LABEL(end) )
```

For the last part of this question, you are to consider **two different translator implementations**, used to translate the following (Mini)Java statement:

```
if (flag>10) result=doThis(); else result=doThat();
```

d) [2pts] Write plausible IR produced by a **translator implementation that uses the classes Nx, Ex and Cx** to generate IR for expressions, specific to the context of their use.

```
SEQ(  CJUMP( GT, TEMP(flag), CONST(10), thn, els),
      LABEL(thn),
      MOVE( TEMP(result), CALL(doThis) ),
      JUMP(NAME(end)),
      LABEL(els),
      MOVE( TEMP(result), CALL(doThat) ),
      LABEL(end) )
```

Key feature of this solution: it contains only ONE CJUMP.

e) [2pts]Write plausible IR, assuming a translator **implementation that translates an AST Expression node directly into an IR Exp node**, and then wraps some glue code around it to fit it into the context of use.

```
SEQ(  CJUMP( EQ, ***, CONST(1), thn, els),
      LABEL(thn),
      MOVE( TEMP(result), CALL(doThis) ),
      JUMP(NAME(end)),
      LABEL(els),
      MOVE( TEMP(result), CALL(doThat) ),
      LABEL(end) )
```

where *** is the result of translating "flag>10" into an IRExp node:

```
*** = ESEQ( SEQ(  MOVE( TEMP(rr), CONST(0) ),
                  CJUMP(GT, TEMP(flag), CONST(10), tt, ff),
                  LABEL(tt),
                  MOVE( TEMP(rr), CONST(1) ),
                  LABEL(ff) ),
            TEMP(rr) )
```

Key feature of this solution: it contains TWO CJUMPs.

5. Canonicalization [10pts]

We have seen a technique to transform IR into a more manageable canonical form. The algorithm (amongst others) conceptually implements a set of rewriting rules that lift ESEQ nodes out of expressions and statements.

The rewriting rules in the book (Figure 8.1) are a subset of the rules necessary to eliminate all ESEQs from expressions. Show the right-hand side(s) for each of the following incomplete rules.

Note that **in some cases you may need to write two right-hand sides** depending on whether something commutes (just as in part (3) and (4) of Figure 8.1).

a) MOVE(TEMP t, ESEQ(s, e)) =>

SEQ(s,   MOVE(TEMP t, e)   )

b) EXP(CALL(e_f, ESEQ(s, e_1), e_2)) =>

if e_f and s commute:

SEQ(s, EXP(CALL(e_f, ESEQ(s, e_1), e_2)))

if e_f and s do not commute (or are not known to commute):

SEQ(MOVE(TEMP f, e_f),
         s,
         EXP(CALL(TEMP f, e_1, e_2))   )

APPENDIX: IR Tree Syntax

Items between /*..*/ are comments indicating the purpose of an element. They are not part of the actual syntax. E.g MOVE contains two sub expressions, the first one is the "destination".

```
Exp ::= CONST( int )
      | NAME( Label )
      | TEMP( Temp )
      | BINOP( Op, Exp, Exp)
      | MEM(Exp)
      | CALL( Exp /*fun*/ (, Exp)* /*args*/)
      | ESEQ( Stm, Exp )

Stm ::= MOVE(Exp /*dst*/, Exp /*src*/)
      | EXP(Exp)
      | JUMP(Label)
      | CJUMP(RelOp, Exp, Exp, Label /*thn*/, Label /*els*/)
      | SEQ( Stm, Stm)
      | LABEL( Label )

Op ::= PLUS, MINUS, MUL, DIV, AND, OR, LSHIFT, ...
RelOp ::= EQ, NE, LT, GT, LE, GE, ULT, ULE, ...
Label ::= <IDENTIFIER>
Temp ::= <IDENTIFIER>
```

To make IR code more readable, it is acceptable to use

```
SEQ(s1,
    s2,
    ...,
    sn)
```
as shorthand for
```
SEQ(s1,
    SEQ(s2,
       .
         .
            sn))))
```

It is also acceptable to use shorthand for BINOPs. For example, you can write PLUS(... , ...) instead of BINOP(PLUS, ..., ...).

Use meaningful formatting and indentation conventions to make the tree/nesting structure of the IR clear.

For example:

```
ESEQ( SEQ( MOVE(TEMP(x),
                CONST(0)),
           MOVE(TEMP(y),
                CALL(NAME(foo),  TEMP(x), CONST(5)))),
      TEMP(y))
```