CPSC 411, Fall 2009 – Final Examination

Name: _____

Q1:	10
Q2:	10
Q3:	10
Q4:	10
Q5:	10
Q6:	10
Q7:	10
Q8:	10
	80

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 best answers to such questions are short, concrete and to-the-point. 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 front side 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 also a a function 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. High Level Question [10pts]

a) [4pts] Why do (some) compilers convert code into IR (intermediate representation), instead of converting it directly into assembly language. Give 2 good reasons and explain each briefly.

reason 1:

This makes it possible to reuse the backend of the compiler for different high-level languages (assuming we implement a translator that generates the same IR). The backend does things such as

- optimizations,
- conversion into executable code
- register allocation
- ...

Each of these can be tricky to implement and so being able to reuse it is important if we want to compile multiple languages/

reason 2:

It is also more easy to retarget the compiler to a different architecture. This is because only some parts of the backend implementation are dependent on the target architecture. Thus, the frontend, as well as some portions of the backend (e.g. optimizations performed on IR code) can be reused rather than rewritten from scratch.

b) [6pts] Give two examples of things that a high-level programming language supports but a low-level language does not. Also **explain briefly how these high-level features might be simulated** by an equivalent low level program produced by a compiler.

example 1:

procedures: low level languages typically have "call" and "ret" instruction. But these instruction only take care of the transfer of control from/to a procedure. The use of parameters, local variables and the procedure's return value all have to be simulated using registers and/or the machine stack.

example 2:

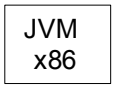
expressions: expressions (with arbitratily nested subexpressions) are a convenience not provided by low-level assembly languages. Expressions have to be simulated by more primitive instructions which leave intermediate results (from evaluating sub-expressions) either on the stack or in registers.

3. Tombstone Diagrams [10pts]

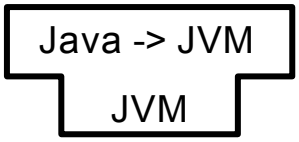
You are given a standard Java SDK (downloaded from Sun). The SDK provides a bytecode compiler called **javac** in the form of a number of .class files containing Java bytecode. It also provides a JVM implementation, in the form of a single **java.exe** binary file that can be run directly on your x86-based windows box. You are also provided with the source code of a Java compiler called **myjavac**. This compiler takes a Java program (i.e. a number of .java source files) and compiles it directly to an x86 .exe file. **Myjavac** is itself implemented in the Java language.

a) Draw tombstone diagram components for all of the given language processors (4 pts)

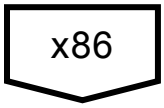
java.exe:



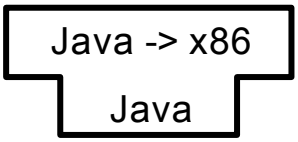
javac:



x86 processor:



myjavac (source code):



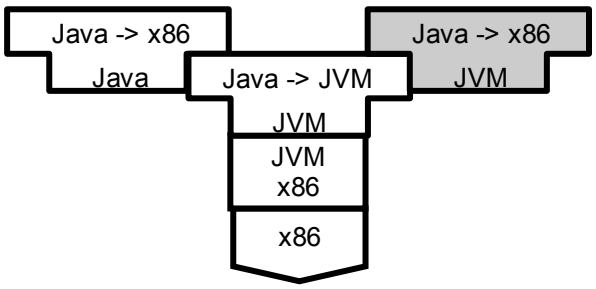
b) Draw a series of tombstone diagrams showing how you can bootstrap the myjavac compiler, using the Sun compiler and JVM. This should end with a diagram showing myjavac running natively on the x86 machine, compiling itself into native code. (6 pts)

1) Compile myjavac into JVM with javac

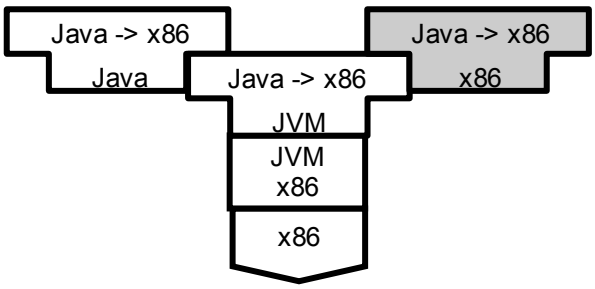
2) Compile myjavac into x86 using compiled myjavac from step 1

3) We now have a version of myjavac running natively on x86, capable of compiling itself

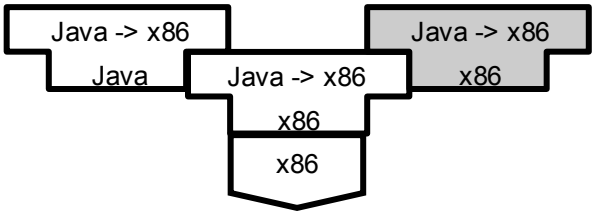
1)



2)



3)



4. Parsing (High Level Question) [10pts]

a) [5pts] You are going to implement a parser, and you will use a parser generator (like JavaCC or SableCC). Your first job is to decide whether you are going to use a top-down (LL) versus a bottom-up (LR) parser tool. What are the advantages and disadvantages of the two different types of parser generators that you should take into account?

Bottom-up:

+ More expressive: this means that we do not have to change the grammar as much to make it acceptable to the algorithm. This in turn has the benefit that our parse tree is likely more intuitive since it corresponds more closely to the syntactic structure we had in mind.

- The algorithms use by a bottom-up parser are more complex. We may not care about this as users of the parser generator (since after all we do not implement the generator). However this complexity has some negative implications for us users:

- Larger parsing tables require more memory

- The generated code is hard to understand and debug. If the parser does not work as it is supposed to, it may be hard to figure out the problem.

- Error messages are harder to understand, and problems are generally harder to fix because of this.

Top-down:

- Less powerful: require more "massaging" of the grammar to make it acceptable to the tool. As a result this may make our parse-tree's less intuitive (or require extra work to make the parser create an intutive parse tree by writing more complex semantic actions).

+ The algorithms used by top-down parser are more intuitive, since they correspond closely to how one might write a parser by hand. This means that if we get error messages it is usually easier to understand why they happen. It is also easier to debug the generated parsers if they don't work as intended (e.g. by reading the generated code and/or stepping it with a debugger.

b) [5pts] Assuming that the language you are implementing is a Scheme-like language (i.e. it uses a LISP-style parenthesized syntax for all its language constructs). What kind of parser generator tool would you use?

Circle one: Top-Down / Bottom-Up

Explain why:

An s-exp grammar is very easy to parse. It would be relatively straighforward to write parser for it by hand. Consequently it would seem like a good choice to use a top-down parser. Though it is less expressive than a bottom-up tool, this lack of expressiveness not be an issue. Althoug we could use a bottom-up tool, it's extra complexity would not really give us any practical benefit for such a simple grammar.

5. Parsing (Implementation) [10pts]

The following is an excerpt from a JavaCC specification for a simple language very similar to MiniJava. For readability and space, it is given in EBNF notation rather than JavaCC syntax.

```
Exp     ::= Term ( "-" Term )*
Term    ::= Prim
          | Prim "." "length"
          | Prim "." <IDE> "(" Exp ( "," Exp )* ")"
          | Prim "[" Exp "]"
Prim    ::= <IDE>
          | <NUM>
          | "(" Exp ")"
```

When this grammar is fed to JavaCC it produces the following error message:

```
Warning: Choice conflict involving expansions at
      ...all  the lines inside of the Term ::= rule ...
      A common prefix is: "(" <IDE>
      Consider using a lookahead of 3 or more for earlier expansion.
```

a) [3pts]What causes this error message?

Essentially, the parser must make a choice between one of the four Term "branches". It uses a lookahead of one token but since all four branches start with Prim, they can all start with the same set of tokens.

As a result the parser can not choose which of the four branches to parse.

b) [3pts]JavaCC suggests using a lookahead. Will using LOOKAHEAD(3) solve the problem? Explain!

No, because Prim can contain an Exp (in between parentesis). As such it can be arbitrarily complex and long. No finite number of lookahead tokens will suffice to "see past" an arbitrarily long Prim.

c) [4pts] Whether or not LOOKAHEAD(3) works here, we can fix the problem by changing the grammar. Write a transformed version of the Term rule to do this:

```
Term  ::= Prim ( "." ( "length"
                     | <IDE> "(" Exp ( "," Exp)* ")"
                     )
                | "[" Exp "]"
                ) ?
```

6. Semantic Checking (Type Checking) [10pts]

Several stages in the compiler check whether an input program is "acceptable" according to the rules of the language. For example, the parser will produce an error when we forget a closing "}".

Each of these errors are static: they are detected at compile time. However, we call the errors detected by the parser "syntax errors" whereas the errors detected by the type checker are called "semantic errors".

a) [3pts] Give an example of a program that has static error that is a semantic (so not a syntactic) error.

boolean x;
x = 0;

Error: x is declared as a boolean but then an integer is assigned to it.

b) [4pts] Explain why it would be difficult for a parser to check for the semantic error in your example above.

Because detecting the error depends on information from the context.

To be more specific: if we consider a statement like "x = 0;" by itself, we can not tell whether it is "incorrect". It may be correct, if x is declared in the context as an int variable.

Parsers are based on "context free grammars". CFG formalism can not specify context dependent constraints on the allowable program structure.

Another complication is that often the "context" may include declarations the parser has not yet read (e.g. in Java method declarations can come in any order).

c) [3pts] Provide a concise yet precise description/definition of what makes a semantic error different from a syntactic error (i.e. how do you know whether some error is semantic rather than syntactic).

Syntax errors are limited to errors that can be detected because the input is not in accordance with the grammar (CFG) of the language. As such, any error which requires context sensitive checking is a semantic rather syntactic error.

7. Type Checking (Implementation) [10pts]

When you implemented your MiniJava type checker, there were essentially three different "architectures" you could choose from.

Architecture 1: Use the **visitor** pattern to traverse and check each node in the AST.

```
class TypeCheck extends Visitor {
   private SymbolTable env;
   @Override class Type visit(Plus n) {
      check(n.e1, new IntegerType());
      check(n.e2, new IntegerType());
      n.setType(new IntegerType());
      return n.getType();
   }
   @Override public Type visit(Times n) {
      check(n.e1, new IntegerType());
      check(n.e2, new IntegerType());
      n.setType(new IntegerType());
      return n.getType();
   }
   ...
}
```

Architecture 2: Add "check" **methods in the AST classes**. Each AST class gets a check method which is responsible to check a node and recursively calls the check method on the children of that node.

```
public abstract class Expression extends AST {
      public abstract Type check(SymbolTable env);
}
public class Plus extends Expression {
   public Type check(SymbolTable env) { ... }
}
public class Times extends Expression {
   public Type check(SymbolTable env) { ... }
}
```

Architecture 3: Use neither a visitor or add methods to the AST classes. Instead, implement your own "dispatch" function **using instanceof**:

```
class TypeCheck {
   private SymbolTable env;
   /** dispatch based on type of exp */
   public Type checkExp(Expression exp) {
     if (exp instanceof Plus)
         return checkPlus((Plus)exp);
     else if (exp instanceof Times)
         return checkTimes((Times)exp);
     else if ...
}
```

Each of these architectures has their own advantages and disadvantages. Choose the architecture that you are most familiar with (presumably because it is the one you chose for your project). On the next page briefly compare your choice with the other two in terms of their relative advantages and disadvantages.

My architecture of choice is: Visitor / AST methods / instanceof (circle one)

a) Compared to: Visitor / AST methods / instanceof (circle a different one)

My choice is better because: (give at least one good reason below)

We can keep the type-checker in a "module" separate from the AST classes. This will likely make the type-checker easier to understand or modify, compared to an implementation of a type-checker that is spread around methods scattered around all the AST classes.

My choice is worse because: (give at least one good reason below)

The visitor pattern tends to result in messier "imperative code" because it is harder to pass information to a visitor from the context of a call. This is because a visitor is intended to be reusable for many kinds of purposes, as such it often does not even accept an argument (other than the visited node).

Visitors tend therefore to have to pass context sensititive information (such as symbol tables representing the environment) by do-ing and un-doing side-effects in the visitor. This is more error prone and cumbersome than simple argument passing.

b) Compared to: Visitor / AST method / instanceof (circle a different one)

My choice is better because: (give at least one good reason below)

Instance-of tests allow us to keep our type-checker as a separate module (separate from the AST classes), yet it still allows us an easy way to pass arguments to the methods that implement the type-checker. Thus we can write our checker in much more functional style with far less side-effects (i.e. we are bot being *forced* to use "do/undo"side effects to remember context sensitive information. We can use either functional parameters / return values or side-effects, whichever seems to fit the situation best. This results in cleaner code with fewer side-effects.

My choice is worse because: (give at least one good reason below)

Instance-of tests are consider "bad OO style". They make the code look ugly with huge if statements and type casts. Moreover, the programming language/compiler has no way of knowing that we might be forgetting case, or can not implement this kind of "if then else" dispatching efficiently if there are many branches.

8. 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+2;
```

```
MOVE( TEMP(sum),
        BINOP(PLUS, TEMP(sum), CONST(2)))
```

b) [2pts]

```
flag = x < y ;
```

```
SEQ(            MOVE( TEMP(flag), CONST(1)),
                CJUMP( LT, TEMP(x), TEMP(y), T, F),
        LABEL(F),
                MOVE( TEMP(flag), CONST(0)),
        LABEL(T))
```

c) [2pts]

```
flag = flag && x < y ;
```

```
SEQ(            CJUMP(EQ, TEMP(flag), CONST(0), Nxt, T),
        LABEL(nxt),
                MOVE( TEMP(flag), CONST(1)),
                CJUMP( LT, TEMP(x), TEMP(y), T, F),
        LABEL(F),
                MOVE( TEMP(flag), CONST(0)),
        LABEL(T))
```

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

```
if (x < y) min = x; else min = y;
```

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( LT, TEMP(x), TEMP(y), T, F),
    LABEL(F),
        MOVE( TEMP(min), TEMP(y)),
        JUMP(NAME(Done)),
    LABEL(T)),
        MOVE( TEMP(min), TEMP(x)),
        JUMP(NAME(Done)),
    LABEL(Done)
```

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), T, F),
    LABEL(F),
        MOVE( TEMP(min), TEMP(y)),
        JUMP(NAME(Done)),
    LABEL(T)),
        MOVE( TEMP(min), TEMP(x)),
    LABEL(Done)
```

Where *** is the result of translating (x < y) into an IR Exp.

```
*** =  ESEQ( SEQ(   MOVE( TEMP(tmp), CONST(1)),
                    CJUMP( LT, TEMP(x), TEMP(y), T1, F1),
                    LABEL(F1),
                    MOVE( TEMP(tmp), CONST(0)),
                    LABEL(T1))
            TEMP(tmp))
```

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,*" should be taken to mean "a list of expressions separated by commas".

```
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,
    s3,
    ...,
    sn)
```

as shorthand for

```
SEQ(s1,
    SEQ(s2,
        SEQ(s3,
            .
                .
                    sn))))
```

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))
```