CS 5363, Fall 2013
Final Examination Review

1 Format

1. Multiple choice / matching questions / short answer (20-50%)
   - recall of vocabulary
   - recognition/recall/application of concepts
   - applications of techniques to come up with an answer, then choose it from available choices
   - differ in format from homework problems
   - Homework problems covering vocabulary/conceptual material: hwk2; hwk3 (#1-3, 6); hwk9; hwk10; hwk11 (#1-5, 7); hwk13 (#2-4)
2. Worked Problems (50-80%)
   - similar in format to homework problems
   - likely to be at least one problem from each of the following categories:
     (a) syntax
       – Homework problems: hwk3 (#4-5); hwk4; hwk5
     (b) context-sensitive analysis
       – Homework problems: hwk6; hwk7
       – also symbol table and type checking (including in compiler project) and identifying declarations based on static scoping
     (c) code generation (tree walk, ILOC, activation records and prologue/precall/postcall/epilogue code)
       – Homework problems: hwk8, hwk11 (#6)
     (d) program analysis (SSA and dataflow analysis)
       – Homework problems: hwk12, hwk13 (#1)
     (e) program optimization (Register Allocation)
       – Homework problems: hwk13 (#5-6)

2 Objectives

2.1 Introductory Material

2.1.1 Background: recursion, dynamic dispatch, and object-oriented tree data structures

1. trace the execution of code using dynamic dispatches.
2. trace the execution of code using recursive function calls.
3. explain how recursion and dynamic dispatch can be used to create, and copy an expression abstract syntax tree.

2.1.2 Compilation and Interpretation as Execution Strategies

1. edit, compile, and run a program from the command-line.
2. edit, and run a program with an interpreter from the command-line.
3. interact a read-eval-print-loop (REPL).
4. explain the difference between an interpreter and a compiler and the benefits of each
5. explain how the standard Java Virtual Machine behaves like an interpreter.
6. explain how the standard Java implementation utilizes compilers and interpreters
7. explain how many language implementations actually combine aspects of compilation and interpretation
8. compare the Java language implementation (source code, the Java source compiler, the Java bytecode representation, the Java Virtual Machine, JIT compiler, interpreter) with the pure interpreter and pure compiler models
9. explain why, technically, a language isn't 'compiled' or 'interpreted'
10. identify language characteristics typically associated with the language being interpreted

2.1.3 Compiler Structure

1. identify the parts of a typical 3-phase optimizing compiler and how those relate to the parts that will need to be implemented for the compiler project.

2.2 Context Free Grammars

1. given a BNF grammar be able to derive sentences using that grammar
2. given a BNF grammar and a (simple) sentence from the language described by that grammar, diagram the parse tree for that sentence
3. given the derivation of a sentence using a BNF grammar, be able to determine whether the sentence is a left-most derivation, a right-most derivation, or neither
4. given a BNF grammar and a parse tree, write the corresponding right-most (or left-most) derivation
5. given an ambiguous BNF grammar and a (simple) sentence that has multiple rightmost derivations from the language described by that grammar, diagram multiple parses tree for that sentence
6. given a BNF grammar for an expression language, determine whether one rule represents an operation of higher precedence than another
7. given a BNF grammar for an expression language, determine whether a rule represents a left-associative or right-associative operation
8. in simple cases, be able to describe in words the language recognized by a BNF grammar
9. in simple cases, be able to write a grammar that recognizes a language described in English
10. in simple cases, be able to give multiple grammars describing a language specified as a grammar or described in English

2.3 Recursive Descent Parsing and LL(1) Languages

1. given an LL(1) grammar for a language, be able to trace how a recursive descent parser would parse a particular sentence in the language
2. given an LL(1) grammar for a language, be able to write a recursive descent parser for that language in pseudocode or a high-level programming language
3. given an LL(1) grammar, be able to identify the FIRST, FOLLOW, and FIRST+ sets
4. given a BNF, be able to determine whether it is LL(1) by computing FIRST, FOLLOW, and FIRST+ sets
5. given a suitable non-LL(1) grammar for an LL(1) language, convert it to be an LL(1) grammar by eliminating left recursion and left-factoring the grammar

2.4 Scanners, Regular Languages, and Finite State Automata

1. formalize lexical descriptions using regular expressions
2. find an NFA equivalent to any given regular expression
3. find a DFA equivalent to any NFA
4. write code implementing a DFA

2.5 Attribute Grammars

1. Given a simple attributed grammar and a sentence in the language described by the grammar, calculate the attributes.
2. Given an attributed grammar, identify attributes of non-terminals for which the attribute is always inherited
3. Given an attributed grammar, identify attributes of non-terminals for which the attribute is always synthesized
4. Given an expression grammar in which the productions for left-associative operations have undergone recursion elimination to produce an LL(1) grammar, be able to write an L-attribute grammar that evaluates the expression applying left-associativity.
5. write an L-attributed grammar that generates an AST for a typical procedural programming language described by a BNF and informal semantics
6. given an L-attributed grammar, write a recursive descent parser with ad-hoc rules implementing the attributed grammar
7. write a tree grammar to type-check the AST of programs written in a typical procedural programming language described by a BNF, informal semantics, and informal type rules

2.6 Type Systems

1. given a language with a set of type-inference rules, and a simple well-typed program, write down a proof-tree showing that the program is well-typed.
2. design and implment a symbol table that can maintain the information needed to type-check a procedural program with declared variables.

2.7 Three Address Code, ILOC, and Control Flow Graphs

1. explain what three-address code is and why it is used in compilers
2. be able to trace the execution of ILOC code
3. be able to explain the meaning of various ILOC instructions
4. given imperative code for a procedure, write equivalent ILOC code
5. given ILOC code for a procedure, draw a control flow graph for the procedure

2.8 Tree Walk Code Generation

1. implement the tree-walk algorithm to translate from AST expression subtrees into ILOC
2. implement the tree-walk algorithm to translate from AST subtrees for while and if statements into ILOC
3. implement the tree-walk algorithm to translate from AST's of simple imperative languages into ILOC

2.9 Variables, Static/Dynamic Scoping, Memory Layout

1. describe the difference between static and dynamic scoping.
2. explain the circumstances under which a variables value can be stored in various locations
3. given a program in a C or Pascal-like language, simulate its execution assuming either static and dynamic scoping.
4. describe how a mark and sweep garbage collector works.
5. simulate the application of the mark pass of a mark and sweep garbage collector to a representation of a program heap in order to determine which objects are reachable and which are unreachable garbage.
6. differentiate the static behavior of a compiler from the dynamic behavior of a compiled program.

2.10 Procedures, etc.

2.10.1 Procedure/Function Values

1. explain what a function value is and why it might be useful
2. trace the execution code that uses procedures/functions as (first- and second-class) values (assuming static scoping)

2.10.2 Parameter Passing Semantics

1. describe the difference among call-by-value, call-by-name, and call-by-reference parameter passing
2. given a program in a C- or Pascal-like language, simulate its execution assuming either call-by-value, call-by-name, and call-by-reference

2.10.3 Implementing Procedures with Activation Records

1. list and describe the contents of a typical activation record
   (a) control and access links
   (b) parameters (call-by-name and call-by-reference)
   (c) temporary values and variable values stored in activation records
   (d) references to objects/structures/arrays on the heap and the objects/structure/arrays themselves
2. trace the execution of function calls in a C or Pascal-like language and be able to diagram a snapshot of the relevant in-memory-structures (as specified in the problem) at a particular point in the program execution including:
   (a) the activation record pointer
   (b) stack of activation records including activation record contents
   (c) closures (pointer to code + pointer to activation record enclosing function value creation) for second-class function

2.10.4 Tail-calls / Tail-recursion optimization

1. describe how tail-call optimization works
2. determine whether a call in a C- or pascal-like language is a tail call

2.10.5 Dataflow Analysis

1. explain how data flow analyses reach a fixed point
2. simulate an iterative dataflow algorithm described by its dataflow equations (not necessarily Dom or LiveOut)

2.10.6 Static Single Assignment Form

1. determine the set of dominators, set of strict dominators, immediate dominator, and dominance frontier of a node in a control flow graph
2. find the dominator tree for a control flow graph
3. explain how SSA serves as a factored use-def chain for efficient global optimizations
4. translate a small program in three-address code into SSA
5. translate a small program (not exhibiting critical edges or mutually dependent ϕ-functions) from SSA into normal three-address code
6. identify critical edges in a control flow graph **(new)**

2.10.7 Register Allocation

1. explain what the difference is between a memory-to-memory model or a register-to-register model is and how using one or the other effects the register allocation problem (i.e., what role does register allocation play in the overall compilation process?)
2. describe the overall plan of the Chaitin-Briggs type register allocator (see Figure 13.10)
   (a) describe how spilling and splitting reduce can be used to make a graph k-colorable
   (b) describe what coalescing is and why it useful
   (c) describe the role played by spill metrics
   (d) describe the trade-offs in reserving or not reserving registers for spilling
3. simulate the execution of a Chaitin-Briggs register allocator
   (a) apply the definition of live range to identify definitions/uses of a variable that must be included in a live range with any particular definition (13.4.1, 13.5.1)
   (b) determine the spill cost of each live range according to some given rule (13.5.2)
   (c) given some ILOC code, build an interference graph (13.5.3)
   (d) simulate the bottom-up coloring of the interference graph (13.5.5)
4. implement register allocation in a compiler (extension – not on exam)
   (a) write a Chaitin-Briggs (Bottom-up global register allocator, 13.5.5) for a TL13 compiler (extension)

3 Vocabulary and Key Concepts

Be able to define, describe, compare, contrast, understand, and use the following vocabulary:

Introductory Material. abstract syntax tree **(new)**, compiler, interpreter, read-eval-print loop, front-end, back-end, 'compiled' language, 'interpreted' language, virtual machine, just-in-time compiler, source language, source code, target language, bytecode, machine code, optimizer, compiler phase

Context-Free Grammars language, grammar, context free grammars, Backus-Naur Form (BNF), derivation, parse-trees, ambiguity, expressions, operator precedence and associativity

Recursive Descent Parsing and LL(1) Languages LL(1) grammars, LL(1) languages, FIRST set, FOLLOW set, FIRST+ set, top-down parsing, predictive parsing, recursive descent parsing, back-track, backtrack-free, packrat parsers **(new)**, memoization **(new)**, parsing expression grammar (PEG) **(new)**, LL(k) grammars, LL(k) languages, ϵ (to represent the absence of symbols in productions), abstract syntax tree **(new)**

Regular Languages, Automata, and Scanners deterministic finite state automata (DFA), nondeterministic finite state automata (NFA), state, transition, ϵ-transition, regular expression, ϵ (to represent the absence of symbols in RE's), alternation/union, concatenation, Kleene closure, configuration of an NFA Thompson's Construction, Subset Construction, DFA Minimization, Kleene's Construction, scanner, ϵ-closure, worklist algorithm

Attribute Grammars attribute grammar, attribute, synthesized attribute, inherited attribute, L-attributed grammar, S-attributed grammar, tree grammar, attributed tree grammar

Types type system, type rule, type safety, type inference premise, conclusion, meta-variable

Three Address Code, ILOC, and Control Flow Graphs three address code, operation, operator/opcode/mnemonic, operand, source operand, destination operand, control flow graph, quadruple

Tree-walk Code Generation Depth-first search, pre-order, in-order, post-order

Variables, Static/Dynamic Scoping, Memory Layout dynamic scoping, static (= lexical) scoping, memory address, base address, offset, heap memory (segment), stack segment, static segment, text (code) segment memory hierarchy, register, cache, main memory, disk storage, garbage collection, stop-the-world (batch) vs. incremental vs. concurrent, mark-and-sweep vs. copying vs. reference counting

Procedures, etc. function value, first-class function values, higher-order functions formal parameter, actual parameter, call-by-name (= pass-by-name), call-by-value (= pass-by-value), call-by-reference (= pass-by-reference), calling convention, precall, prologue, epilogue, postcall, activation record (= call stack frame), call stack, activation record pointer (= frame pointer), control link (=dynamic link), access link (=static link), closure, tail call, tail call optimization (= tail call elimination), dynamic dispatch

Dataflow Analysis live (variable/value/register), gen set, kill set, liveout, dataflow problem, iterative data-flow analysis, forward/backward flow problem, fixed-point, monotone function

Static Single Assignment Form static single assignment form, ϕ-functions, dominate, dominance frontier, immediate dominator, minimal SSA, pruned SSA, semi-pruned SSA, join node, critical edge

Register Allocation register assignment vs. register allocation, physical register vs. virtual register, register-to-register model vs. memory-to-memory model, local vs. global register allocation, live range, to spill a register, spill code, spill cost, interference graph, graph coloring, register class, coalescing live ranges, and to split a live range