CS 322 Languages and Compiler Design II — Final

NAME:

EMAIL:

Check here if you have used a cribsheet: ☐ (Cribsheets will NOT be used for grading)

Instructions:

- Do not turn over this page until the test begins.
- Time allowed is 100 minutes (ONE hour and FORTY minutes).
- You may bring a single cribsheet in to the exam room to use during this test PROVIDED that it is: no larger than a sheet of letter size paper, handwritten, writing on only one and one half sides, and turned in with your answers at the end of the test. No other materials, including textbooks, calculators, notes, computers, etc. may be used.
- You may choose to leave the exam at any time during the first 85 minutes; please be sure to hand in your work before leaving, and exit quietly to avoid disturbing other students. To avoid unnecessary disruption, please do not leave the exam in the final 15 minutes of the exam.
- The papers that you turn in should be clearly labeled with your name and email address.
- Please do all your work on these sheets. Try to keep your answers as clear, as concise, and as legible as possible. None of these problems requires very long answers.
- The points for each section are shown in parentheses on the right-hand side. There are four questions on this paper, each of which is worth a total of 25 points. Credit will be given for partial answers, and for correct working or explanation, even if the final answer is incorrect. If you make any assumptions about the interpretation of a question, be sure to document that in your solution.
- Answer as many questions or parts of questions as you can. By design, you will probably NOT have enough time to answer all of the questions; the intention is that you will have plenty of questions to choose from!
- Beware of trick questions!
- You may ask for clarification of questions at any time during the exam.
- Discussion of this exam with colleagues or classmates is not allowed until all solutions have been turned in.
- Breaches of academic integrity will be treated very seriously.

1) **Static versus Dynamic, and Formal Specifications:**

a) Explain, in general terms, how the terms *static* and *dynamic* are used when talking about programming language implementations. (2)

static refers to aspects of an implementation that are handled at compile-time.

dynamic refers to aspects of an implementation that are handled at run-time.

Briefly compare/define each of the following pairs of terms, indicating how the general concepts of *static* and *dynamic* apply in each case.

– Static versus dynamic semantics. (2)

static semantics = aspects of behavior (e.g., types) that are determined at compile time

dynamic semantics = behavior of program at run-time.

– Static versus dynamic linking for building executable programs. (2)

static linking = building executable program by combining object files + library routines before runtime.

dynamic linking = code for library routines is loaded & linked into running code.

– Static versus dynamic links in stack frames. (2)

static link = ptr to stack frame of lexically enclosing function (determined by syntactic struct. of program)

dynamic link = ptr to stack frame of calling function (determined by execution of program).

The Java programming language includes a `static` keyword that can be used as part of a method declaration, but it does not include a keyword for `dynamic` methods. Drawing on patterns in the preceding examples, explain what is meant by a *static* method in Java, and comment on what you would expect if Java included support for *dynamic* methods. (2)

static method: a "class" method, invoked without a "this" object, function being called is known at compile time.

dynamic method = virtual method, address of function to be called is determined at runtime by type of "this".

b) Describe what is meant by a *formal* description of a programming language, and explain why it might be useful to develop such a description for a language like Java or C++. (4)

- A formal description of a programming language is an attempt to provide a complete, unambiguous, and precise characterization of the language in terms of some well-understood mathematical model.
- A formal description of C++ or Java would provide a basis for:
  - ensuring consistent behavior of diff. lang. implementations.
  - establishing safety properties of untrusted code (e.g., a Java web applet)
  - rigorous study of algorithm correctness

c) Briefly describe three different techniques that can be used to give a formal description of the meaning of programs written in a simple imperative programming language. (6)

operational semantics: uses rules describing behavior of an abstract machine to characterize the behavior of source program fragments.

axiomatic semantics: uses logical formulae involving program variables, together with inference rules, to specify program behavior.

denotational semantics: describes the behavior of programs using functions that map syntactic structures to values (or "denotations") in some underlying semantic domain / mathematical model.

Continues on next page ...

d) To illustrate your answer to Part (c) in more depth, pick ONE of the three techniques that you have listed and describe how it might be used to formalize the runtime behavior of loops of the form "`do S while(E);`" where `S` can be any statement, and `E` is a Boolean-valued expression. Be sure to define or explain any special terms or notation that you use in formulating your answer. (5)

Different answers possible depending on choice.

e.g., axiomatic:

$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q \wedge E\}\ S\ \{Q\}}{\{P\}\ \underline{\text{do}}\ S\ \underline{\text{while}}\ E\ \{Q\}}$$

Q = invariant

/3 Notation, plausible rule

/2 Details

etc...

2) **Interpreters and For Loops:**

a) Suppose that we are given an interpreter, *Int*, and a compiler, *Comp*, for a programming language L. Explain, as precisely as you can, what you would expect each of these software components to do, and how you would expect them to be related, both in function and in consruction. **(6)**

An interpreter executes programs and can be thought of as a function mapping programs to meanings:

$$Int_L : L \to M$$

A compiler translates programs from one language to another & can be thought of as a function:

$$Comp : L \to L'$$

Relationship in function: $\forall p \in L.\ Int_L\, p = Int_{L'}\,(comp\, p)$

In construction: both require source input, lexical analysis, parsing, static analysis components, which could be shared between implementations of Int, Comp.

b) What are the most common design goals and characteristics of interpreters in *practical systems* that distinguish them from traditional compiler systems? [Note: There is space to continue your answer to this question on the top of the next page.] **(5)**

- Interpreters often designed w/ more emphasis on interactive use (e.g., education, rapid prototyping, ...) [cf. batch mode of typical compilers]
- Interpreters often built w/ less emphasis on raw performance.

Continues on next page ...

- Interpreters can often be written in a more portable manner without worrying (so much) about specifics of host machine architecture.
- Interpreters can be used to specify prog. lang. semantics, or as a platform for prog. lang. research.

c) Describe what is meant by a bytecode interpreter, and explain why this approach might be used to build practical interpreter-based system. **(5)**

- A bytecode interpreter is an interpreter for an idealized abstract machine language in which programs are represented as sequences of byte values.
- A practical interpreter might include an internal bytecode compiler that translates source programs into equivalent bytecode to be executed by a bytecode interpreter. This can help to reduce interpretive overhead, enable more compact representation, and maintain portability.

d) Assuming either a bytecode system or else direct interpretation over abstract syntax (for example, using `exec()` and `eval()` methods), describe how you would add support for a for-loop construct of the form:

```
for (init; test; incr) stmt
```

to an interpreter for a C/C++/Java-like language. You should assume that `init` and `incr` are arbitrarily-typed (and optional) expressions, `test` is an (optional) Boolean-valued expression, and `stmt` is an arbitrary statement. You can, of course, assume the existence of code for handling these individual parts of the for-loop in an appropriate way, but your answer should include brief descriptions of any assumptions that you make. **(9)**

[Write your answer on the next page ⟹]

[space for answers continues here]

Assume abstract syntax for a for-loop with fields:

init, test, incr (optional expressions)
stmt (a program stmt)

With direct interpretation, we could define the following exec() method:

method: /2
general logic: /3
optional parts: /3
exit from body: /1

```
Value exec(Memory mem) {
    if (init ≠ null) {
        init.eval(mem);
    }
    while (test==null || test.eval(mem).isTrue()) {
        val = stmt.exec(mem);
        if (val ≠ null) {
            return val;
        }
        if (incr ≠ null) {
            incr.eval(mem);
        }
    }
    return null;
}
```

[Incidental detail: returns either "null" if loop terminates, or else value that is produced by an enclosed return inside the stmt.]

Continues on next page ...

3) **Garbage Collection and Runtime Representation:**

a) Describe two schemes for implementing dynamic memory allocation, one that involves incrementing a heap pointer, while the other involves allocating from a free list. **(4)**

- if memory is collected in a single contiguous block; maintain a heap pointer that points to first free/unused location; allocate by increasing pointer & checking for overflow. /2
- if free memory is distributed across multiple disjoint blocks: maintain a linked list of free regions; allocate by searching for a block of the appropriate size then removing that block (or, at least, the allocated portion) from the list. /2

b) What are the advantages and disadvantages of using garbage collection in comparison to a system that requires explicit deallocation of memory? **(2)**

+ gc saves programmer work (no need to write explicit free/destroy code)
+ gc avoids bugs with premature deallocation of live data
- can be harder to control space usage
- may result in execution pauses when heap becomes full.

c) Outline the main steps in an algorithm for garbage collecting the heap in an implementation of an object-oriented programming language like Java where objects can come in different sizes, and include both pointers to other objects as well as non-pointer fields such as integer or character values. **(10)**

- General details

  identify roots
  - global variables
  - pointers into heap in current stack frames or registers.

  /2

  visit objects
  - needs some method to identify embedded pointer fields; may also need size/tag/copying info.

  /3

- Specifics

  Outline description of mark-sweep
  or two-space
  or other collector

  (See slides...)

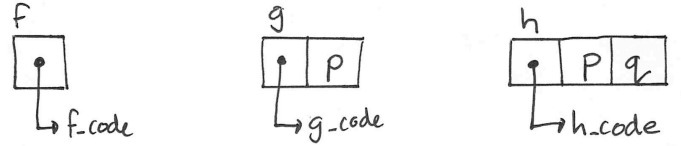
  /5

Continues on next page ...

d) Write `intfun` as a name for the type of functions that take a single `int` value as an argument and return an `int` value as a result. The following code fragment shows how values of this type might be used in a simple test program:

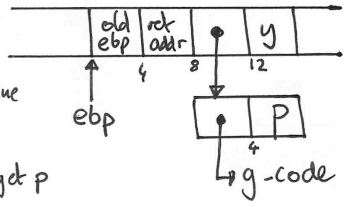
```
int test(int p, int q) {
  int at0(intfun f) { return f(0);     }
  int f(int x)      { return x + 1;    }
  int g(int y)      { return y + p;    }
  int h(int z)      { return p*z + q; }
  return at0(f) + at0(g) + at0(h);
}
```

Draw diagrams to show a likely representation (i.e., memory layout) in an x86 computer for closures corresponding to the definitions of `f`, `g`, and `h` (each of which is a value of the `intfun` type) in the above code. Your answer should include the code that will be pointed to by the closure for `g`, but you do not need to provide code for either `f` or `h`. Do NOT use/assume any form of function inlining. **(6)**



Each closure begins with a pointer to code for the closure, followed by values of free vars. /3

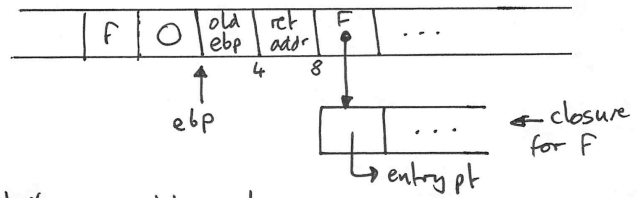
```
g_code:
  pushl  ebp          ┐
  movl   esp, ebp     ┘ prologue
  movl   8(ebp), eax  ┐
  movl   4(eax), eax  ┘ get p
  addl   12(ebp), eax   add y
  movl   ebp, esp     ┐
  popl   ebp          ┘ epilogue
  ret
```



/3

could use space here too... ☺

e) Sketch a possible implementation (in IA32 assembly code, and following the usual calling conventions) for the `apply()` function defined in Part (c). **(3)**




```
at0:  pushl  ebp
      movl   esp, ebp
      pushl  $0
      movl   8(%ebp), eax
      pushl  eax
      movl   (eax), eax
      call   *eax
      movl   ebp, esp
      popl   ebp
      ret
```


Continues on next page ...

4) **Optimization Techniques:**

a) What are the most important goals and limitations in the construction of an optimizer for a practical compiler? [Simply naming goals and limitations will not receive full credit, so be sure to include brief explanations as well in each case!] **(4)**

Goals:
- optimization should preserve the semantics of the original program
- optimization should improve the performance of the original program (wrt. time, space, ...)

Limitations:
- can't make up for poor algorithms/data structures.
- may require tradeoffs between goals (e.g. time vs space).
- may require expensive analysis.
- true optimization is not computable.

b) Explain what is meant by a *peephole optimization*, and give three examples for the IA32 architecture to show how peephole optimizations can be used to improve the quality of generated machine code. **(4)**

A peephole optimization rewrites a short sequence of (assembly language) instructions with an equiv., more efficient sequence.

```
e.g.  addl  $1, %eax        ] ⇒ incl %eax
      imull $2, %eax        ] ⇒ addl %eax, %eax
      addl  $12, %esp       ]
      movl  %ebp, %esp      ] ⇒ movl %ebp, %esp
```

c) Explain carefully what is involved in each of the following different types of optimization, and give a simple example in each case to illustrate how the optimization works: (8)

- Code motion: moves code from one part of a program to another (e.g., out of loop body) to avoid repeated comp.

e.g., `for (i=0; i<n*m; i++){` ⇝ `int L = n*m;`
`..code that doesn't change n or m...` ⇝ `for (i=0; i<L; i++){`
`}` `...` `}`

- Common subexpression elimination: replaces code that recomputes a value with code that reuses the previously computed value:

e.g., $x = a + b$, $y = a + b$ ⇝ $x = a + b$, $y = x$

- Strength reduction: replaces code for an expensive operation with code for a cheaper but equivalent operation.

e.g., $x = y^2$ ⇝ $x = y * y$
$x = 16y$ ⇝ $x = y << 4$

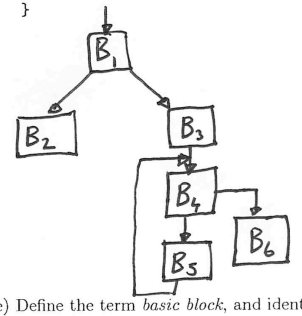
- Dead code elimination: removes code that will not be executed or that has no effect.

e.g., $x = y + t$, $x = t + 1$ ⇝ $x = t + 1$

Continues on next page ...

d) Write down a translation of the following Java/C/C++ procedure into three-address code. DO NOT attempt to optimize the generated code IN ANY WAY at this stage. (3)

```
int fastFib(int m) {
  int f0 = 0, f1 = 1, f2, i;
  if (m<=1) {
     return m;
  } else {
     for (i=2; i<=m; i++) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
     }
     return f2;
  }
}
```



B_1 [1) f0 = 0
2) f1 = 1
3) if m>1 then 5]
B_2 [4) return m]
B_3 [5) i = 2]
B_4 [6) if i>m then 12]
B_5 [7) f2 = f0 + f1
8) f0 = f1
9) f1 = f2
10) i = i+1
11) goto 6]
B_6 [12) return f2]

three-addr /1
plausible translation /2

e) Define the term *basic block*, and identify the basic blocks in the program fragment from Part (d). Include a brief sketch to show the structure of the underlying flowgraph. (3)

A basic block is a sequence of instructions that always begins at the first instruction and exits at the last instruction. (i.e., no entries/exits mid-block) /1

Basic blocks above: 1-3, 4, 5, 6, 7-11, 12 /1

Flow graph above /1

f) Explain what is meant by a *global* optimization, and describe any opportunities for this kind of optimization that are revealed by translating the `fastFib()` function into three-address code. What other techniques might be used to ensure that a compiler generates an efficient machine code implementation for this function? **(3)**

A "global optimization" is an optimization that involves code in multiple basic blocks. /1

Initialization of f0, f1 (instrs 1, 2) can be moved into block B_3 because they are not used until B_5. [example of code motion.] /1

Other techniques: register allocation
or instruction scheduling
or ... /1

Continues on next page ...

[This page may be used for rough working and notes.]

End