Questions to ask yourself in preparation for tests and quizzes

Mayer Goldberg

October 21, 2017

Contents

1 Introduction ................................................. 2
   1.1 How to use this document ............................... 2
   1.2 How not to use this document .......................... 3
   1.3 How I will respond to questions about this document 3

2 Abstraction .................................................. 4

3 Scheme programming ....................................... 4
   3.1 Sexprs .................................................. 4
   3.2 The quote-form ......................................... 5
   3.3 The backquote mechanism ............................... 6
   3.4 The void object ........................................ 6
   3.5 The empty list (nil) .................................... 6
   3.6 Numbers ............................................... 7
   3.7 Booleans ............................................... 7
   3.8 Characters ............................................ 7
   3.9 Pairs & Lists .......................................... 8
   3.10 Strings ............................................... 8
   3.11 Vectors ............................................... 9
   3.12 Symbols ............................................. 9
   3.13 Logical operators .................................... 10
   3.14 Procedures .......................................... 11
   3.15 let-forms ........................................... 11
   3.16 Side effects .......................................... 11
   3.17 Files & IO ............................................ 11
   3.18 System ............................................... 12

4 Static vs Dynamic, Early Binding vs Late Binding 12

5 The functional & imperative paradigms .................. 13

6 Functional programming .................................. 13

7 Interpretation & compilation ............................. 14

8 Dynamic & lexical scope .................................. 15
9 Closures & methods

10 Parameter-passing mechanisms

11 Syntax

12 Macros & syntactic sugar

13 Fixed points & fixed-point combinators

14 The tail-position, tail-recursion optimization, & tail-call optimization

15 Continuation-Passing Style (CPS)

16 The CISC architecture (given in cisc.h)

17 Code generation

18 Compiler optimizations

19 Run-time support

20 Dynamic memory management

1 Introduction

1.1 How to use this document

This document contains questions for you to think about. If you think hard about these questions, they will

• Help you understand better the material taught in the course.

• Help you recall important topics & issues discussed in class and in the recitation sessions.

• Train you to make the connections between various topics discussed in the course, as well as between these topics and your everyday work as a programmer.

• Give you practice in thinking critically & creatively about the course material.

• Prepare you for the midterm & the final exam.

The best way to go over this document is in pairs: Find a good study partner, someone at about your own level, and work out answers to these questions by alternating with them: You should think about each question separately, and then discuss your answers with each other, trying to convince, explain, illustrate your thoughts. When your partner is explaining their answer to you, try to find holes in their argument: Be adversarial, come up with counter-examples, and challenge them at each step. Then it shall be your turn to take this treatment from your partner...

It is not important if you fail to solve all the questions. It is important, however, that you attempt each and every one, and that you apply yourself sincerely to coming up with original, precise, clever answers. Some of these questions are open-ended. Some have more than one possible answer. Don’t worry about that! If you and your study partner follow the regiment described herein, you will learn a lot regardless of whether you end up in agreement or in disagreement.
1.2 How not to use this document

I would like you to think of this document as a “mental gym” for the compiler-construction course. The questions are “exercise equipment”, and to benefit from them, you must do the exercises! If you follow this analogy, then it is clear how not to use this document: You should avoid shortcuts at all costs:

- I am quite certain that some kind soul will immediately begin compiling “correct answers” to these questions. I have no way of preventing them from doing this, and I can only encourage you not use such answers: Often there is more than one way of looking at the question. If you buy into someone else’ answers, you will have denied yourself the opportunity to come up with your own answers. You cannot get into shape by letting someone else exercise for you, you cannot raise your IQ by eating gifted children, and you cannot learn anything by starting with a solution manual someone else prepared for this document. Please don’t be clever about this admonition, and don’t tell yourself that you’ll only use the pre-compiled answers to check your own… This will get you into lazy mental patterns, and remove any healthy tensions that should build up as you are working through the problems and questions.

- Studying alone will be less effective, because you will not be as critical. You need a study partner to disagree with you, to challenge you, and to force you to thinking more clearly. This means that you will probably only get to answer at most half the questions in this document. Hopefully the document will continue to grow and evolve, so that half of it will still give you a worthwhile exercise.

- Studying in groups larger than two will be less effective, because even if everything else goes right, and you’re all super-motivated, you will each be answering less than half the questions in total... Why lose that many questions? Also, interaction is more complicated in larger groups: There will be more people pushing for breaks, more people who discuss other things, etc.

- Don’t get stressed out if you can’t figure out all the answers:
  - I’m not going to cover all this material each year. Just practice on those problems that are relevant to the current year.
  - Many of these questions are phrased in an open-ended way. Most are open, many have more than one answer.
  - Some require you to make a value judgment. For such questions, there is no singularly, objectively, unique “correct” answer. There are only aspects, considerations, and context: Define your own, narrow down the question, and form your own judgment. Be prepared to defend it before your study partner!

1.3 How I will respond to questions about this document

Just as I am certain that some do-gooder will compile a list of answers, so I am certain that students will be asking me for the answers to these questions, or at least “to discuss these questions” with them. If you follow the analogy of the document being a “mental gym”, you can see why I am reluctant to lift your own weights for you.

So here is what I am willing to do: You are always welcome to come and see me, during office hours, and tell me what YOU think about some of these questions. I’ll then be happy to raise my own objections, just as your study partner ought to have done... Please don’t ask me to do
more, because by doing so, you will only be depriving yourself of an opportunity to do some mental exercise. You **will not improve** just by listening to me and my opinions... You have to do the mental work yourself!

## 2 Abstraction

1. Give examples of functional abstraction.
2. Give examples of object-oriented abstraction.
3. Give examples of abstraction in logic.
4. Give examples of abstraction in mathematics.
5. Give examples of textual abstraction.
6. Give example of *operators* or *higher-order functions* in mathematics.
7. Give examples where abstraction is incorrect.
8. Give examples where abstraction offers no advantages.
12. Give an example of modeling functional abstraction using textual abstraction.
13. Give an example of modeling functional abstraction using abstraction in logic.
14. Give an example of modeling object-oriented abstraction using abstraction in logic.
15. Discuss the difference between static and dynamic abstraction.
16. Give examples of meaningfully abstracting different elements of the same code.

## 3 Scheme programming

### 3.1 Sexprs

1. What does the term *sexpr* stand for?
2. What is *symbolic computation*?
3. Give examples of *symbolic computation*.
4. What are the advantages of using *sexpr*s?
5. What are the disadvantages of using *sexpr*s?
6. What are the issues involved in constant, static data?
7. What is *RTTI* and why is it related to *sexpr*s?
8. What are the advantages of RTTI?

9. What are the disadvantages of RTTI?

10. What is the difference between an integer in Scheme and an integer in C?

11. What is the difference between a character in Scheme, a primitive char in Java, a byte in Java, and an instance of the Character class in Java?

12. Discuss the contention that Scheme is type-impoverished.

13. Suppose a language such as C might want to include sexprs:
   (a) Would you recommend this?
   (b) What would be the implications for the type system in C?

14. Discuss the anomaly of quote.

15. Write code in Scheme that demonstrates the anomaly of quote.

16. Write code in C that demonstrates the anomaly of quote.

17. What is the approach taken with respect to the anomaly of quote in Dr Racket?

18. How is the anomaly of quote related to the operating system on which the code is running?

19. How is the anomaly of quote related to the computer architecture on which the code is running?

20. Is the anomaly of quote likely to show up in a Scheme running on DOS 3.2 running on an Intel 80386 micro-architecture?

21. What is necessary for the anomaly of quote to show up in a Scheme running on DOS 3.2 running on an Intel 80386 micro-architecture?

22. Discuss the implications of the anomaly of quote to the backquote mechanism.

23. Describe the two dot rules that govern how sexprs are printed.

3.2 The quote-form

1. What constant sexprs must be preceded with a quote?

2. What is the value of a quoted expr, if the original expr needs no quote?

3. What does it mean to say that `<sexpr>` is a reader macro?

4. Discuss the reason why Prolog has no need for a quote form.

5. Discuss the implications of eliminating the quote form (“are constants really necessary?”).
3.3 The backquote mechanism

1. What is the reader-macro for unquote?

2. What is the reader-macro for unquote-splicing?

3. Give an example in which an unquote-splicing-expression can be modeled using an unquote-expression.

4. Give an example in which an unquote-splicing-expression can not be modeled using an unquote-expression.

5. Give an example in which an unquote-expression can be modeled using an unquote-splicing-expression.

6. Can all unquote-expressions be modeled using unquote-splicing-expressions?

7. Does the backquote mechanism always result in a dynamic expression?

8. At what stage in the compiler pipeline is the expansion of backquote handled?

9. Write an expander for a one-level backquote mechanism.

10. Write an expander for the nested version of the backquote mechanism.

11. Discuss the difference, in terms of complexity of the grammar, between support for the one-level backquote mechanism, and the nested version of the backquote mechanism.

3.4 The void object

1. Discuss the motivation for having the void object.

2. Concerning situations in Scheme where the void object is returned, what happens in corresponding situations in other programming language that do not have a void object?

3. How does one test for the void object?

4. How does one create the void object?

5. What forms in Scheme make use of the void object?

6. When is the void object printed and when is it not printed?

7. Why is there no characteristic predicate for the void object?

3.5 The empty list (nil)

1. What is the characteristic predicate for nil?

2. Can we test for nil by comparing an object against the nil object?

3. Is nil a pair?

4. Write a Scheme procedure that takes an sexpr and returns the number of occurrences of nil.
3.6 Numbers
1. What is the characteristic predicate for numbers?
2. What is the characteristic predicate for integers?
3. What is the characteristic predicate for complex numbers?
4. What are the differences between an integer in Scheme and an \texttt{int} in C?
5. What are the differences between an integer in Scheme and an instance of \texttt{Integer} in Java?
6. Discuss numerical overflow & underflow in C.
7. Discuss numerical overflow & underflow in Java primitive types.
8. Discuss numerical overflow & underflow in Java objects of type \texttt{Integer} & \texttt{Float}.
9. Discuss numerical overflow & underflow in Scheme.
10. Is the \texttt{less-than} (\texttt{<}) predicate polymorphic in Scheme?
11. Is the \texttt{less-than} operator polymorphic in Java?
12. Compare the numerical tower in Scheme to the numerical tower in C.
13. Compare the numerical tower in Scheme to the numerical tower in Java.
14. Discuss auto-conversion within the numerical tower in Scheme.
15. Compare auto-conversion within the numerical tower in C & Java.

3.7 Booleans
1. What is the characteristic predicate for Booleans in Scheme?
2. What is the \texttt{negation} of a non-Boolean?

3.8 Characters
1. What is the characteristic predicate for characters in Scheme?
2. How to test for uppercase & lowercase chars?
3. How to convert chars to uppercase & to lowercase?
4. How to compare characters?
5. How to compare characters regardless of case?
6. How to convert an integer to a char?
7. How to convert a char to an integer?
8. What are \texttt{named chars}?
9. What \texttt{named chars} are there?
3.9 Pairs & Lists

1. Lists in Scheme and Prolog are defined *inductively* using the *empty list* and *pairs*. Lists in other languages (e.g., Python) are not defined inductively. Explain the implications of the inductive definition of lists.

2. Define *proper list*.

3. Define *improper list*.

4. Is *nil* a *proper list*?

5. Discuss the differences between comparing pairs using `eq?` and comparing pairs using `equal?`.

6. Discuss the difference between lists and vectors.

7. In your Scheme programming (PPL, and the compiler construction course) you have not had much occasion to use *ordered pairs* that were not lists.
   
   (a) Discuss the advantages of using *ordered pairs* that are not lists.
   
   (b) Discuss the disadvantages of using *ordered pairs* that are not lists.

8. Give an example of a *proper list* that is printed with a dot.

3.10 Strings

1. What is the characteristic predicate for strings in Scheme?

2. Discuss the difference between finding the length of a string, and finding the length of a list.

3. When comparing two strings using `eq?`, what is the meaning of the result?

4. Discuss mutability of strings in Scheme.

5. Discuss the differences between strings in Scheme and in C. Your answer should not be limited to the issue of *RTTI*.

6. Discuss the differences between strings in Java and in C. Your answer should not be limited to the issue of *RTTI*.

7. What information is contained in the *null string* (also known as the *empty string*) in Scheme?

8. How are strings compared?

9. What is the complexity of the *equality predicate* on strings?

10. Compare strings in Scheme & Java.

11. Compare strings in Scheme & C.

12. Compare constant strings in Scheme & Java.

13. How to find the length of a string in Scheme?

14. What is the complexity of finding the length of a string in Scheme?
15. How are strings catenated in Scheme?
16. What is a string meta-character?
17. What are the string meta-characters in Scheme?
18. How to specify characters in hex or in octal within a string?
19. How to create a string from a list of characters?
20. How to create a list of characters from a string?
21. How to convert strings to uppercase & to lowercase?
22. Write code to check whether a given string is a palindrome.

### 3.11 Vectors

1. What is the characteristic predicate for vectors in Scheme?
2. Discuss the *size* difference between a vector of *n* elements, and a list of *n* elements.
3. Discuss the difference between recursive programming over a vector, and recursive programming over a list.
4. What information is contained in a vector of length 0 in Scheme, and an array of length 0 in C.
5. Discuss mutability of vectors in Scheme.
6. Discuss the difference between recursion on *pairs* and recursion on *vectors*.

### 3.12 Symbols

1. Discuss the difference between symbols & strings.
2. What is the *print name* of a symbol?
3. How are symbols compared?
4. When comparing two symbols with `eq?`, what is the meaning of the result?
5. How to convert a string to a symbol?
6. How to convert a symbol to a string?
7. Discuss the notion of the *length of a symbol*.
8. How are symbols represented in standard Scheme?
9. How are symbols represented in our compiler?
10. Discuss the notion of catenation of symbols.
11. What are *interned symbols*?
12. What are *uninterned symbols*?
13. What is returned by the `gensym` procedure?

14. How is it that symbols generated through `gensym` are not equal (in the sense of `eq`) to symbols with the exact same print name?

15. What makes it possible for symbols to be compared in constant time irrespective of the number of characters in their print names?

16. What is 2-LISP?

17. Discuss the contention that the programming language Perl is an example of 5-LISP.

3.13 Logical operators

1. Compare `and` in Scheme with `&&` in Java.

2. Compare `or` in Scheme with `||` in Java.

3. Compare `and` in Scheme with `&&` in C.

4. Compare `or` in Scheme with `||` in C.

5. Explain why `and` and `or` are special forms, while `not` is a procedure.

6. Can `and` be defined using `or`, `not` and de Morgan’s laws?

7. Can `or` be defined using `and`, `not` and de Morgan’s laws?

8. Can `and` or `or` be defined using `nand` or `nor`?

9. Should `nand` be implemented as a procedure or special form?

10. Should `nor` be implemented as a procedure or special form?

11. In Scheme, the special forms `and` & `or` may take expressions the values of which aren’t Boolean, while in Java, the corresponding `&&` & `||` take only expressions the values of which are Boolean.

   (a) Discuss the advantages & disadvantages of the approaches taken by Scheme & Java.

   (b) Describe, using UML class diagrams, how you might implement `sexpr` classes in Java, so that the behaviour of `and` & `or` would be captured.

12. What would be required in order to implement `and` & `or` as `procedures` rather than as `special forms`?

13. What are the varieties in the syntax of `if`-expressions?

14. What must be the value of the `test`-expression of an `if`-expression in Scheme?

15. Compare the value of the `test`-expression of an `if`-expression in Scheme to corresponding expressions in C & Java.

16. Compare the `if`-expression in Scheme to the `conditional form` in C & C++.

17. What kinds of nested `if`-expressions can be rewritten using a single `cond`-expression?
3.14 Procedures
1. What is the characteristic predicate for procedures in Scheme?
2. Does the characteristic predicate for procedures distinguish between user-defined procedures and built-in procedures?
3. What is the value of the top-level variable +?
4. Define a closure.
5. When is the body of a procedure evaluated?

3.15 let-forms
1. Define the ribs of let-forms.
2. Which rib-bindings can see each other in a let-expression?
3. Which rib-bindings can see each other in a let*-expression?
4. Which rib-bindings can see each other in a letrec-expression?

3.16 Side effects
1. Define side effect.
2. Define the notion of environment with respect to side effects (hint: This has nothing to do with the lexical environment or with the global environment discussed later on).
3. What is the difference between a statement and an expression?
4. Which special forms perform side-effects, and on what?
5. Which procedures perform side-effects, and on what?
6. In general, what is the notational convention for naming procedures that perform side-effects?
7. What procedures, that perform side-effects, do not abide by the notational convention mentioned in question #6?
8. What architectural features/aspects of the code are broken by the general use of side-effects?
9. Side effects often require syntactic support for sequences of statements. What kinds of sequences are there in Scheme?

3.17 Files & IO
1. How to convert sexprs to strings?
2. How to read an sexpr from stdin?
3. How to read an sexpr from an input port?
4. How to write an sexpr to stdout?
5. How to write an *sexpr* to an *output port*?

6. How to read a character from *stdin*?

7. How to read a character from an *input port*?

8. How to write a character to *stdout*?

9. How to write a character to an *output port*?

10. How to test for the *end-of-file* object?

11. How to close input & output ports?

12. How to open input & output ports?

13. How to check whether a file exists?

14. How to delete a file?

### 3.18 System

1. How to exit the Scheme session?

2. How to *evaluate* an *sexpr*?

3. How to issue an error message?

4. How to run a shell command from within Scheme?

### 4 Static vs Dynamic, Early Binding vs Late Binding

1. Define the term *static*, especially as it is used in the field of programming languages.

2. Define the term *dynamic*, especially as it is used in the field of programming languages.

3. What are the advantages of using static programming languages?

4. What are the disadvantages of using static programming languages?

5. What are the advantages of using dynamic programming languages?

6. What are the disadvantages of using dynamic programming languages?

7. List some language-features that are unique to static programming languages.

8. List some language-features that are unique to dynamic programming languages.
5 The functional & imperative paradigms

1. Define the environment.
2. Of what can the environment consist?
3. What does it mean for a programming language to have no side-effects? If it has no side effects, how do we ever see the outcome of the computation?
4. How can the assignment-statement/expression be modeled in purely-functional programming?
5. How can the goto-statement be modeled in purely-functional programming?
6. How can the print-statement be modeled in purely-functional programming?
7. How can a pseudo-random number generator be modeled in purely-functional programming?
8. How can gensym be modeled in purely-functional programming?
9. How can a file be modeled in purely-functional programming?
10. How can a 80×25 text screen/window be modeled in purely-functional programming?
11. How can the terminal buzzer or beeper or bell be modeled in purely-functional programming?
12. How can event-driven programming be modeled in purely-functional programming?

6 Functional programming

1. Define functional programming.
2. What is a thunk?
3. What does it mean to thaw a thunk?
4. Define Currying.
5. What is the relation between a Curried function and a corresponding un-Curried function?
6. What is map?
7. What is the value of a call to map, with some function, over nil?
8. Give an example of using map with a function of two arguments.
9. Give an example of using map with a variadic procedure.
10. When is the composition of maps not equal to the map of the composition?
11. What is the procedure filter?
12. Write your own version of the procedure map for variadic procedures.
13. Write your own version of the procedure filter.
14. What is the procedure andmap?
15. What is the procedure `ormap`?
16. What is the result of applying `andmap` to a unary function and `nil`?
17. What is the result of applying `ormap` to a unary function and `nil`?
18. Give an example of using `andmap` with a variadic procedure.
19. Give an example of using `ormap` with a variadic procedure.
20. Write your own version of `andmap` for variadic procedures.
21. Write your own version of `ormap` for variadic procedures.
22. What is defunctionalization?
23. What functions don’t lend themselves easily to defunctionalisation?
24. What is the stack-machine transformation?
25. Convert the direct-style version of `fact` (the factorial function) to a stack machine.
26. Convert the direct-style version of `fib` (the Fibonacci function) to a stack machine.
27. Convert the direct-style version of `ack` (the Ackermann function) to a stack machine.
28. Convert the direct-style version of `length` (on lists) to a stack machine.

7 Interpretation & compilation

1. Define interpretation.
2. Define compilation.
3. State the relationship between interpreters & compilers.
4. What is the most-basic & fundamental interpreter, on which all code must run?
5. Discuss the logical necessity (for computation) of compilers & interpreters.
6. Characterize virtual machines (VMs) in terms of compilers & interpreters.
7. Define de-compilation.
8. Define cross-compilation.
10. Define compiler optimizations.
11. What does it mean for a compiler to be correct?
12. What does it mean for a compiler-optimization to be correct?
13. Discuss the relationship between compiler optimizations and abstraction in programming languages.
15. Give examples of stacking interpreters.

8 Dynamic & lexical scope

1. What does the general term dynamic mean in programming languages?
2. What does the general term static mean in programming languages?
3. What is meant by the expression “the scope of a variable”?
4. Define dynamic scope.
5. Define lexical scope.
7. Describe deep binding dynamic scope.
8. Describe shallow binding dynamic scope.
9. State some additional dynamic features that deep binding has, and that shallow binding has not.
10. Give examples of expressions that have the same value under lexical & dynamic scope.
11. Give examples of expressions that have different values under lexical & dynamic scope.
12. Discuss what it means for a piece of code “to be correct” under dynamic scope.
13. Recall the implementation of dynamic scope in the PPL course: Would you characterize it as deep binding, shallow binding, or something entirely different?
14. Discuss the relationship between exception handling and dynamic scope.
15. Discuss exception handling in languages such as C++ & Java, and its relationship to dynamic scope.
16. Which form of dynamic scope is implied by exception handling: shallow binding or deep binding?
17. What restrictions would you place on the use of exception handling in C++ & Java, in order to catch exceptions efficiently?
18. Discuss the implications of dynamic scope for closures.
19. Define lexical address.
20. Discuss the differences between the lexical address implied by the interpreter in PPL vs the lexical address used in the compiler construction course.

\footnote{You didn’t, in fact, compute lexical addresses in PPL, however this could have been done, and the question pertains to the to this possibility.}
21. In PPL, you did not separate the initial/global environment from the lexical environments constructed during the evaluation of Scheme code. Assuming you were to separate the initial/global environment from the lexical environment, what kinds of variables would there be in the Scheme interpreter you implemented in PPL?

22. What kinds of variables are there in the compiler you are writing in the compiler construction course?

23. Define the procedure `occurs?`, that takes a variable name v, and a parsed expression pe, and returns #t if pe contains an occurrence of the variable v, or #f, otherwise.

24. Is there a difference between how you would define `occurs?` for PPL and for the compiler-construction course?

25. Define the procedure `occurs-free?`, that takes a variable name v, and a parsed expression pe, and returns #t if pe contains a free occurrence of v, or #f, otherwise.

26. Is there a difference between how you would define `occurs-free?` for PPL and for the compiler-construction course?

27. Define the procedure `occurs-bound?`, that takes a variable name v, and a parsed expression pe, and returns #t if pe contains a bound occurrence of v, or #f, otherwise.

28. Is there a difference between how you would define `occurs-bound?` for PPL and for the compiler-construction course?

29. Define the procedure `occurs-param?`, that takes a variable name v, and a parsed expression pe, and returns #t if pe contains an occurrence of a parameter of v, or #f, otherwise.

30. Define the procedure `vars`, that takes a parsed expression pe, and returns a set of all variables that occur in pe. The point of the result being a set, is that the result should contain at most one occurrence of each variable name, i.e., no duplicates.

31. Is there a difference between how you would define `vars` for PPL and for the compiler-construction course?

32. Define the procedure `free-vars`, that takes a parsed expression pe, and returns a set of all variables that occur freely in pe.

33. Is there a difference between how you would define `free-vars` for PPL and for the compiler-construction course?

34. Define the procedure `param-vars`, that takes a parsed expression pe, and returns a set of all parameters that occur in pe.

35. Is there a difference between how you would define `param-vars` for PPL and for the compiler-construction course?

36. Define the procedure `bound-vars`, that takes a parsed expression pe, and returns a set of all variables that occur bound in pe.

37. Is there a difference between how you would define `bound-vars` for PPL and for the compiler-construction course?
38. Is it possible for a variable name to have occurrences as a free variable, bound variable, and parameter, all in a single expression?

39. Is it possible for a variable to occur as a bound variable but not as a parameter, all in a single expression?

40. Define a lexical addressing algorithm for the lexical environments implied by the interpreter in PPL. Remember to separate free/global variables from other kinds of variables.

41. Describe the lexical addressing algorithm you were given for the compiler.

42. Suppose that rather than macro-expanding the let-forms (let, let*, and letrec), you were to support them directly from the abstract syntax tree and all the way to the code generator: Modify the lexical addressing algorithm to cover the let-forms.

43. In the interpreter you wrote in PPL, when is the lexical environment extended?

44. In the compiler you are writing in the compiler-construction course, when is the lexical environment extended?

45. Discuss the effects of extending the lexical environment in different code points (PPL vs compiler-construction course) on the space & time efficiency of closure creation, set & get (for parameters, bound variables, and free variables), and procedure application.

9 Closures & methods

1. Compare procedures in Scheme and in C.

2. Compare procedures in Scheme and methods in Java.

3. What is the structure of a closure.

4. What is the structure of an object in a class-based object-system?

5. What is the structure of an object in a classless object-system?

6. Give an example of 4 closures that all share the same lexical environment, but not the same code.

7. Give an example of 4 closures that all share the same code, but not the same lexical environment.

8. Give an example of 4 objects, instances of different classes, that all share the same non-static state.

9. Give an example of 4 objects that all share the same methods.

10. What concept in object-oriented languages corresponds to bound variables in functional languages?

11. What concept in object-oriented languages corresponds to parameters to a closure in functional languages?
12. What concept in object-oriented languages corresponds to the *lexical environment* in functional languages?

13. In functional languages, when different closures share the same *code pointer* but each has a different *environment*, how does the code “know” how to access the *bound variables* in the correct environment?

14. In object-oriented languages, when different objects share the same method (at the same address), how does the method code “know” how to access the *instance variables* in the correct instance/object?

15. What is the structure of the activation frame when evaluating a closure application?

16. What is the structure of the activation frame when evaluating a method call?

### 10 Parameter-passing mechanisms

1. Define *call-by-value*.

2. Define *call-by-reference*.

3. Define *call-by-sharing* (aka *call-by-object*).

4. Define *call-by-name*.

5. Define *call-by-need*.

6. Discuss the differences between *call-by-reference* and *call-by-sharing*.

7. What expressions can be passed by reference?

8. A variable is passed *by value*. Will an assignment to this variable in the *callee* be seen by the *caller*?

9. A variable is passed *by reference*. Will an assignment to this variable in the *callee* be seen by the *caller*?

10. A variable is passed *by sharing*. Will an assignment to this variable in the *callee* be seen by the *caller*?

11. A variable is passed *by name*. Will an assignment to this variable in the *callee* be seen by the *caller*?

12. A variable is passed *by need*. Will an assignment to this variable in the *callee* be seen by the *caller*?

13. What is the *Jensen device*?

14. When is it possible to *set* a parameter passed *by name*?

15. What happens when attempting to set a parameter passed *by name* that is unsuitable to be set? Is this a *compile-time* issue or a *run-time* issue?

16. Discuss the contention that parameter-passing mechanisms should be specified in the *caller*, rather than the *callee*. 
17. Give an example of a call where it is more efficient to pass the argument *by need* rather than *by name*.

18. Give an example of a call where it would be incorrect to pass the argument *by need* rather than *by name*.

19. Give an example of a call where it would be more efficient to pass the argument *by value* rather than *by name*.

20. Give an example of a call where it would be more efficient to pass the argument *by object* rather than *by value*.

11 Syntax

1. Define *concrete syntax*.

2. Define *abstract syntax*.

3. Define *token*.

4. Define *delimiter*.

5. Define *left-recursion*.

6. Define an *ε-production*.

7. Explain how *ε-productions* are eliminated.

8. Explain how *left-recursion* is eliminated.

9. What stage in the pipeline of the compiler is concerned with moving from *concrete syntax* to *abstract syntax*?

10. Define *top-down* parsing.

11. Define *bottom-up* parsing.

12. Define *recursive-descent* parsing.

13. Define *parsing combinators*.

14. What form of parsing is captured by *parsing combinators* (top-down or bottom-up)?

15. Discuss the problems posed to top-down parsers by *ε-productions*.

16. Discuss the problems posed to top-down parsers by *left-recursion*.

17. Discuss the advantages of *parsing combinators*.

18. Discuss the disadvantages of *parsing combinators*.

19. What grammatical features cause *parsing combinators* to *diverge* (enter an *infinite loop*)?

20. Discuss the reasons for moving from *concrete syntax* to *abstract syntax*.
21. Is it possible to perform semantic operations on programs (interpretation, compilation, semantic analysis, optimization, etc) directly on the concrete syntax?

22. What kinds of errors or problems in the source code are generally uncovered while going from concrete syntax to abstract syntax?

23. What kinds of errors or problems in the source code are generally uncovered while examining the abstract syntax tree?

24. The prefix #; comments-out an entire sexpr. How is it different from line comments in terms of the pipeline of the compiler?

12 Macros & syntactic sugar

1. Define syntactic sugar.

2. Discuss the advantages of using syntactic sugar in the user code.

3. Discuss the disadvantages of using syntactic sugar in the user code.

4. Discuss the advantages of syntactic sugar to the compiler-writer.

5. Discuss the disadvantages of syntactic sugar to the compiler-writer.

6. Give a macro-expansion for let.

7. Give several macro-expansions for let*.

8. Give several macro-expansions for letrec.

9. Give several macro-expansions for begin.

10. Explain why we do not use the macro-expansion for begin in our compiler.

11. Give several macro-expansions for and.

12. Give several macro-expansions for or.

13. Explain why we do not use the macro-expansion for or in our compiler.


15. Discuss whether apply can be implemented as a macro.


17. Give two examples of macro-expansion for or-expressions that are not lexically hygienic.

18. Give two examples of macro-expansion for begin-expressions that are not lexically hygienic.

19. Discuss the ways in which gensym is related to lexical hygiene.

20. What is the limitation of using gensym to achieve lexical hygiene?

21. Invent syntax for a for-loop in Scheme, and suggest a hygienic macro-expansion for it. If at all possible, avoid the use of gensym.
22. Compare macro-expansion in Scheme with macro-expansion in C.
23. Discuss the disadvantages of macro-expansion in general.
24. Is macro-expansion in C lexically hygienic?
25. Discuss the implications of macros on debugging.

13 Fixed points & fixed-point combinators

This section is based on the standard presentation in class, where for each function $f$, we define $F_f, G_f, H_f$:

- The function $f$ is the least fixed-point of $F_f$.
- The function $G_f$ satisfies $(G_f G_f x) = (f x)$ for all $x \in \text{Dom}(f)$.
- The function $H_f$ satisfies $((H_f H_f) x) = (f x)$ for all $x \in \text{Dom}(f)$, so $(H_f H_f)$ computes $f$.
- When we are talking about a specific $f$, we often drop the subscripts and just refer to the corresponding $F, G, H$ functions for that given function $f$.

1. What is a fixed point of a function.
2. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has no fixed points.
3. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has exactly 1 fixed-point.
4. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has exactly 2 fixed-point.
5. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has exactly 5 fixed-point.
6. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has countably-many fixed-points.
7. Give an example of a function ($\mathbb{R} \rightarrow \mathbb{R}$) that has uncountably-many fixed-points.
8. Why does a higher-order functional (the $F$-functions...) always have a fixed point?
9. What is a least fixed-point?
10. Why does a least fixed-point of a higher-order functional always exist?
11. What is the significance of the notion of leastness in the least fixed-point of a higher-order functional?
12. What is the fixed-point iteration?
13. Describe the relationship between leastness and the fixed-point iteration?
14. Define a recursive version of the normal-order $\text{fix}$ operator for higher-order functionals.
15. Define a recursive version of the applicative-order $\text{fix}$ operator for the higher-order functionals.
16. For several recursive functions, define the corresponding F-functions (higher-order functionals).
   
   (a) Prove [in two parts, the first without using induction and the second, using induction] that the recursive functions are indeed least fixed-points of their corresponding F-functions.
   
   (b) Verify in Scheme that the recursive functions are indeed least fixed-points of their corresponding F-functions.

17. For several recursive functions, define the corresponding G-functions.
   
   (a) Prove [by induction] that the original recursive function can be computed using the corresponding G-function.
   
   (b) For each recursive function, run the corresponding G-function in Scheme and in C, and verify they can be used to compute the recursive function for various inputs.

18. Define circular data-structures using a fixed-point combinator.

19. Define mutually-recursive functions using a circular data-structure and a fixed-point combinator.

20. What is the smallest procedure, the least fixed-point of which is the infinite loop?

21. Write down the recursive, call-by-name, fixed-point operator.

22. Write down the recursive, call-by-value & call-by-sharing fixed-point operator.

23. Write down the call-by-name fixed-point combinator.

24. Explain why the call-by-name fixed-point combinator & operator diverge under call-by-value & call-by-sharing.

25. Write down the call-by-value & call-by-name fixed-point combinator.

26. What is \( \eta \)-reduction?

27. What is \( \eta \)-expansion?

28. Give an example to explain when \( \eta \)-reduction is valid, and why.

29. Explain how \( \eta \)-expansion is used to go from the call-by-name fixed-point combinator & operator to the call-by-value & call-by-sharing fixed-point combinator & operator, respectively.

**14 The tail-position, tail-recursion optimization, & tail-call optimization**

1. How is a tail-call identified?

2. How is a tail-recursive call identified?

3. State the difference between the tail-call optimization and the tail-recursion optimization.

4. State the difference between immediate tail-recursion and non-immediate tail-recursion.
5. What is achieved by performing the \textit{tail-recursion optimization}?

6. What is the significance of the \textit{tail-call optimization} in Scheme?

7. Does the \textit{tail-call optimization} optimize \textit{space}, \textit{time}, or both?

8. What enables the \textit{tail-call} optimization in Prolog?

9. What is the correct behaviour for \texttt{(apply f s)} when this application is in tail-position?

10. What is the correct behaviour for \texttt{(apply f s)} when this application is not in tail-position?

11. Give an example of code that would cause a stack overflow if \texttt{(apply f s)} is not implemented correctly with respect to the tail-call optimization?

12. Discuss the implications of the tail-call optimization for \texttt{and}-expressions.

13. Give an example of a macro-expansion for \texttt{and}-expressions that interferes with the tail-call optimization.

14. Discuss the implications of the tail-call optimization for \texttt{or}-expressions.

15. Give an example of a macro-expansion for \texttt{or}-expressions that interferes with the tail-call optimization.

16. Discuss the implications of the tail-call optimization for \texttt{cond}-expressions.

17. Discuss the implications of the tail-call optimization for \texttt{begin}-expressions.

18. Give an example of a macro-expansion for \texttt{begin}-expressions that interferes with the tail-call optimization.

19. What sub-expressions of a \texttt{let}-expression can never be in \textit{tail-position}?

20. What sub-expressions of a \texttt{cond}-expression can never be in \textit{tail-position}?

21. What sub-expressions of an \texttt{if}-expression can never be in \textit{tail-position}?

22. What sub-expressions of a \texttt{lambda}-expression can never be in \textit{tail-position}?

23. What sub-expressions of a \texttt{define}-expression can never be in \textit{tail-position}?

24. What sub-expressions of an \texttt{application} can never be in \textit{tail-position}?

25. Is it possible to write the predicate \texttt{in-tail?} that would take a Scheme expression, and return \texttt{#t} if all the applications in the given expression are in tail position, or \texttt{#f}, otherwise?

26. Discuss the implications of the tail-call \& tail-recursion optimizations for debugging.

27. You are programming in a language that implements the tail-call optimization. You need to “turn off” this optimization, so that you could debug your code. What is the simplest thing you can do?
15 Continuation-Passing Style (CPS)

1. What is control in programming languages?

2. Give examples of control structures in programming languages.

3. What is a continuation?

4. Discuss the relation between CPS and the tail-position.

5. What is direct style?

6. Discuss how to identify programs in direct style?

7. Discuss the use of CPS with multiple continuations.

8. Discuss the use of CPS where the continuation takes several arguments.

9. Is it an optimization, to convert code from direct style to CPS?

10. Is it possible to write the predicate in-cps? that would take a Scheme expression, and return #t if that expression is written in CPS, or #f, otherwise? (compare with section 14, question #23)

11. When converting code from direct style to CPS, where in the code are new continuations created?

12. How to convert and-expressions to CPS?

13. How to convert or-expressions to CPS?

14. How to convert cond-expressions to CPS?

15. How to convert let-expressions to CPS?

16. How to convert set!-expressions to CPS?

17. How to convert if-expressions to CPS?

18. There’s a problematic situation that may occur when converting if-expressions to CPS. Explain the situation, and explain how to solve it.

19. If, upon converting code to CPS, no new continuations were created, what can we deduce concerning the original code?

20. Discuss the effects of CPS on the order of evaluation of an expression.

21. In proving correctness of the CPS transformation, what is the induction hypothesis?

22. For the standard, recursive factorial, function, and its CPS counterpart, prove the correctness of the CPS version.

23. For the standard, recursive Fibonacci, function, and its CPS counterpart, prove the correctness of the CPS version.

24
24. For the standard, recursive Ackermann, function, and its CPS counterpart, prove the correctness of the CPS version.

25. A program can be converted from recursive to iterative form, when the operator that combines the result of the recursive call is associative. For example, in the case of the factorial function, this operator is multiplication. In the case of the Fibonacci function, this operator is addition. Given that CPS is a general method for converting any recursive to iterative form, what is the associative operator that makes this conversion possible?

16 The CISC architecture (given in cisc.h)

1. Look over cisc.h and note all the extensions to the syntax of standard ANSI C. Explain why these extensions were necessary.

2. How can you specify the maximum RAM size?

3. How can you specify the maximum stack size?

4. How can you add more register?

5. There is precisely one register in the machine defined in cisc.h, that has not been exposed to the programmer. What is it, and what is its purpose?

6. Describe the differences in use & implementation between the PUSH & PUSHA instructions.

7. Describe the differences in use & implementation between the JUMP & JUMPA instructions.

8. Describe the differences in use & implementation between the CALL & CALLA instructions.

9. Characterize the kind of programming that cannot be done with cisc.h.

10. Describe the difference between the frame assumed by the FPARG assembler directive, and the frame assumed by our code generator.

11. Write an assembly routine to implement the gcd function.

12. Write an assembly routine to implement a test for primality (whether an integer argument is a prime number).

13. Write an assembly routine to implement a quicksort on an array of numbers, given the address of the array and its size.

14. What is the significance of assembly address 0. Which library function manipulates it directly?

15. While various data types, architectural sizes, assembler directives, and assembly instructions can be added to the CISC architecture, there is one kind of programming that cannot be practiced directly in CISC. Discuss this kind of programming, what it means to say that it cannot be practiced directly in CISC, and how can it be practiced indirectly.

16. What are the effects of overflow, underflow in our CISC architecture.
17 Code generation

1. What is the *induction hypothesis* for [the proof of correctness of] the compiler?

2. One of the general-purpose registers has a special rôle in the *induction hypothesis* for [the proof of correctness of] the compiler. Which is it, and what is that rôle?

3. Explain why the code generated for a 100-element, constant list, and a single integer are substantially the same.

4. Why does the code-generator generate *indexed* labels?

5. Describe the code generated for applications in tail-position.

6. Describe the code generated for applications that are not in tail-position.

7. Describe the code generated for simple *λ*-expressions.

8. Describe the code generated for *λ*-expressions with optional arguments.

9. Describe the code generated for *set!*-expressions to parameters.

10. Describe the code generated for *set!*-expressions to bound variables.

11. Describe the code generated for *set!*-expressions to free variables.

12. Describe the code generated for *define*-expressions.

13. What is the return value of a *set!*-expression?

14. What is the return value of a *define*-expression?

15. Describe the code generated for *or*-expressions.

16. Describe the code generated for *if*-expressions.

17. Describe the code generated for parameters.

18. Describe the code generated for bound variables.

19. Describe the code generated for free variables.

20. How many words are used to represent an integer?

21. How many words are used to represent a fraction?

22. How many words are used to represent *nil*?

23. How many words are used to represent *void*?

24. How many words are used to represent a Boolean value?

25. How many words are used to represent a character?

26. How many words are used to represent a string?

27. How many words are used to represent a symbol?
28. How many words are used to represent a vector?
29. How many words are used to represent a pair?
30. How many words are used to represent a closure?

18 Compiler optimizations

1. What are high-level optimizations?
2. Give examples of high-level optimizations.
3. What are intermediate-level optimizations?
4. Give examples of intermediate-level optimizations.
5. What are low-level optimizations?
6. Give examples of low-level optimizations.
7. Discuss the special significance of optimizations on loops.
8. Give examples of optimizations on loops.
9. What are peephole optimizations?
10. Give examples of peephole optimizations.
11. Give examples of one optimization enabling another optimization.
12. Give examples of one optimization preventing another optimization.
13. What is the optimization of common sub-expression elimination?
14. Give examples of the optimization of common sub-expression elimination.
15. Give examples of where applying the optimization of common sub-expression elimination would be incorrect, because it would change the value of the expression.
17. Give examples of jump optimizations.
18. Give examples of move optimizations.
19. Give examples where partial inlining of a loop improves the body of the loop.
20. Give examples where partial inlining of a loop does not improve the body of the loop significantly.
21. Give examples where software-pipelining improves the body of the loop.
22. Give examples where software-pipelining does not improve the body of the loop.
23. Give examples of loops in which software-pipelining is not possible.
19 Run-time support

1. Describe several major differences between interactive and non-interactive (offline) compilers.

2. Discuss the contention that interactive systems are interpreted systems.

3. Describe the implementation of the apply procedure.

4. Describe how the apply procedure is related to the tail-call optimization.

5. The procedure map is variadic, takes functions of any arity, followed by the corresponding number of lists of arguments. Implementing map in Scheme will create many intermediate lists unnecessarily. Discuss how an implementation of map in CISC assembly can avoid creating these intermediate lists.

6. The Scheme standard specifies that the behaviour of side-effects on constants is undefined, that is, it is entirely left to the discretion of the implementer. In light of this, how does our compiler handle side-effects on constants?

7. Explain the advantages of implementing variadic functions (such as +, *, etc) in assembly, rather than in Scheme.

8. Define the term read-eval-print-loop (REPL).

9. What is a threaded-code compiler?

10. Discuss the limitations of threaded-code compilers.

11. In an offline Scheme compiler, what is needed to support the read procedure?

20 Dynamic memory management

1. Describe Knuth’s algorithm for allocating and de-allocating chunks of memory.

2. Give an example of two chunks of garbage that point to each other.

3. Describe reference counting.


6. What is the difference between reference counting and garbage collection in terms of eagerness vs laziness?

7. Give an example where reference counting out-performs garbage collection.

8. Give an example where garbage collection out-performs reference counting.

9. Discuss the advantages of mark-and-sweep over stop-and-copy.

10. Discuss the advantages of stop-and-copy over mark-and-sweep.

11. When would you prefer mark-and-sweep over stop-and-copy?
12. Describe the *generational hypothesis*, aka the *infant mortality hypothesis*.

13. Give an example of code that satisfies the *infant mortality hypothesis*.

14. Give an example of code that violates the *infant mortality hypothesis*.

15. Give an example of Scheme code that creates short-lived garbage.

16. Give an example of Scheme code that creates long-lived garbage.

17. Give an example of a short-lived list of long-lived elements.

18. Give an example of Scheme code in which it is essential to garbage-collect closures.

19. Give an example of Scheme code in which it is essential to garbage-collect assembly-language code.

20. Give an example of code for which an eager strategy for memory-management is preferable to a lazy strategy for memory-management.

21. Give an example of code for which a lazy strategy for memory-management is preferable to an eager strategy of memory-management.

22. Give an example of code that creates a data structure, such that, once collected, enables to collect another data structure.

23. Give an example of code that creates data structures that must be collected from the stack (and that are not pointed to from the top-level).

24. Write code in Scheme that causes the garbage-collector to run over and over again.

25. Write code that fills memory with long-lived, small chunks of garbage.

26. Write code that fills memory with short-lived, large chunks of garbage.