cs173: Programming Languages

Final Exam

Fall 2001

Please read all the instructions before you turn the page.

This exam is worth 101 points. We will assign partial credit to partial responses, provided we can decipher the response, it reflects a correct understanding of course material, and does not violate the requirements of the problem. Exam rules:

- Your responses to this exam are due by 1200hrs on 2001-12-15 (Saturday). We will post turn-in instructions on the course newsgroup.
- If you believe a question is underspecified or buggy, make a reasonable assumption, and document your claim and your work-around. (However, we believe none of the questions fall in this category. Therefore, you should invoke this right rarely or never—don't turn in blow-by-blow thoughts on every response!)
- The exam is open book with respect to the course notes for the year 2001 (including Wilson's monograph, notes provided by guest lecturers, and newsgroup postings). You may not refer to any other sources related to the course between the time you start and finish the exam.
- You have three hours and 15 minutes to take this exam. Time begins when you start reading the exam questions. The first three hours must be a single, contiguous block. At some point, either immediately or later (but before the responses are due), you may use an additional 15 contiguous minutes to change or augment your answers.
- All Scheme code responses must use the dialect employed in this course. You may not use assignments or continuations unless expressly permitted to do so in the problem statement.
- Put your name on every single page you turn in.
- Solutions may be hand-written, but the burden of legibility rests on you.
- Unless you are typesetting your entire solution, please fill your answers in a print-out of the exam itself. As much as possible, use the spaces provided for each response. If you use an overflow sheet, number it carefully and indicate its use in the main exam. Staple all sheets together. If you fail to perform any of these steps, we may simply overlook parts of your response.
- You *may* evaluate any of the programs related to this exam on a computer.
- You must neither give assistance to any other course participant, nor receive assistance from anyone other than the course staff, on material under examination.

Brown University has an Academic Code that governs this exam and all our other transactions. I expect you, as students and scholars, to abide by it faithfully and fully.

Problem	Max	Score
1	15	
2	5	
3	20	
4	15	
5	20	
6	5	
7	20	
8	1	
Total	101	

Problem 1: CPS [15 point(s)]

CPS the following function:

```
(define (fold f l acc)
  (cond
   [(empty? l) acc]
   [(cons? l)
    (fold f
          (rest l)
          (f (first l) acc))]))
```

You need not convert the list primitives (*empty?*, *rest*, etc). You should assume that the first argument to *fold* has itself been converted into CPS (and may not be primitive).

Convert into CPS the following use of *fold* so that it uses the CPS-converted version above:

(*fold* + (*list* 1 2 3 4) 0) ;; this evaluates to $1 + 2 + 3 + 4 = 10$

Problem 2: Overloading [5 point(s)]

Many scripting languages include a feature they call “polymorphism”, but is technically known as *overloading*. In this feature, one operator name serves as a front for several different, unrelated operators; the language picks the right operator based on the types of arguments given at run-time. For instance, + implements addition when given numbers, concatenation (string-appending) when given strings, list appending when given lists, and so forth.
Why is overloading different from polymorphism (as we’ve studied it in this class), even though it superficially seems similar?

Problem 3: Type Inference [20 point(s)]

Consider the following typed Rip expression:

$\{proc\ \{f : \square_1\} : \square_2$
$\quad \{proc\ \{x : \square_3\} : \square_4$
$\quad\quad \{proc\ \{y : \square_5\} : \square_6$
$\quad\quad\quad \{cons\ x\ \{f\ \{f\ y\}\}\}\}\}\}$

We have left the types unspecified ($\square_n$) to be filled in by the type inference process.
Derive type constraints from the above program. Then solve these constraints. From these solutions, fill in the values of the boxes. Be sure to show *all* the steps specified by the algorithms (i.e., writing the answer based on intuition or knowledge is insufficient). You should use type variables where necessary.
To save writing, you can annotate each expression with an appropriate type variable, and present the rest of the algorithm in terms of these type variables alone (to avoid having to copy the corresponding expressions). If you do this, be sure to annotate *every sub-expression* with a type variable. Be sure the annotations are clearly readable!

Problem 4: Memory Management and Type Soundness [15 point(s)]

We've said, in class, that the existence of some pointer-oriented operations in languages like C and C++ make it impossible to prove a type soundness theorem for those languages. What primitive(s) cause(s) a problem, and what is that problem?

For a language like Scheme, ML or Java, is garbage collection *necessary* to establish type soundness? Why or why not?

Problem 5: Generational Garbage Collection [20 point(s)]

A generational garbage collector naturally tracks references from newer objects (ones allocated more recently) to older ones. A major challenge is keeping track of references that go the other way: from old objects to new.
What in the design of a copying generational collector makes it "natural" for the collector to track references from new to old, rather than vice versa?

Distinguish between variable mutation and value mutation. In variable mutation, we change the value held by a variable itself. In value mutation, the variable still refers to the same object; it's the content of the object that changes. (For example, **set!** in Scheme implements variable mutation, while *vector-set!* implements value mutation.)
Which of these does a generational collector need to track specially? For each one, state whether it is or isn't tracked, with a brief justification for why or why not.

There is one fundamental *property* (as opposed to a mere implementation detail) common to page-marking, word-marking and card-marking that store lists do not share. What is this?

Problem 6: Conservative Garbage Collection [5 point(s)]

Why does a conservative garbage collector employ a mark-and-sweep strategy (which is generally less efficient than advanced copying strategies)? You may find many reasons, but state the one that is clearly the most important.

Problem 7: Stack Walking [20 point(s)]

Some applications share information between their sub-systems by placing marks on the executing program's stack. Java's security manager functions this way, as does the Stepper in DrScheme. The following two Rip expressions represent a very simple version of these operations:

- {*mark* ⟨*RP*⟩ ⟨*RP*⟩} evaluates the first sub-expression, places its value on the stack as a *mark*, then evaluates its *body*, which is the second sub-expression. When the body finishes evaluating, the mark disappears from the stack. The value of the entire expression is the value of the body.
- {*most-recent-mark*} walks the stack from newest-to-oldest frames, until it finds a mark, and returns this mark as its value.

Using the CPSed interpreter on the next page as your starting point, implement these two operations. (To make the problem simpler, we've removed procedures, variables and applications from the language, leaving only arithmetic.) Assume the programmer never writes an ill-formed mark query, i.e., all {*most-recent-mark*} requests lie within enclosing *mark* expressions.
Here are some examples of program behavior:

```
{mark 4
   3} ⟹ 3
{mark {+ 3 4}
   {+ 5 6}} ⟹ 11
{mark {+ 3 4}
   {most-recent-mark}} ⟹ 7
{mark 4
   {mark 5
      {most-recent-mark}}} ⟹ 5
{mark 4
   {mark {most-recent-mark}
      3}} ⟹ 3
{mark 4
   {mark {most-recent-mark}
      {most-recent-mark}}} ⟹ 4
{mark 4
   {+ {mark 5
         {most-recent-mark}}
      {mark 6
         {most-recent-mark}}}} ⟹ 11
{mark 4
   {+ {mark 10
         1}
      {most-recent-mark}}} ⟹ 5
```

Hint: Previously, a continuation responded to only one kind of "message": consume a value, and perform the rest of the computation. Now, a continuation may want to respond to two different kinds of messages ...

```
;; An RP is one of
(define-struct numE (num))          ;; num : number
(define-struct addE (lhs rhs))      ;; lhs : RP, rhs : RP

(define (interp/k expr k)
  (cond
    [(numE? expr) (k expr)]
    [(addE? expr) (interp/k (addE-lhs expr)
                            (lambda (lhs-v)
                              (interp/k (addE-rhs expr)
                                        (lambda (rhs-v)
                                          (k (numE+ lhs-v rhs-v))))))]
    [else (error 'interp "not a valid expression" expr)]))

(define (run expr)
  (interp/k expr (lambda (x) x)))
```

Problem 8: Bonus Question [1 point(s)]

What programming language would Thomas Jefferson have used?