cs173: Programming Languages

Final Exam

Fall 2002

Please read this page of instructions
before you turn the page!

This exam is worth 102 points. We will assign partial credit to partial responses, provided we can understand the response and it reflects a correct understanding of course material.
Exam rules:

- Your responses to this exam are due at noon on Tuesday, December 17. Please read the newsgroup for instructions on turning in the exam.
- On preparing solutions:
  - Solutions may be hand-written, but the burden of legibility rests on you.
  - Unless impossible, please answer the questions on these exam sheets, and within the space provided. If you do use separate sheets of paper, be sure to make clear where your work on a problem begins and ends.
  - Staple your solutions together. *Put your name on every page.*
- Do not assume that the length of an answer should be proportional to the number of points assigned. Even for weighty problems, the correct solution can take just a few words. No problem on this exam asks for, or even really wants, a lengthy essay as a solution.
- If you believe a question is underspecified, make a reasonable assumption and document it.
- The exam is open book with respect to the course notes for the year 2002. The exam may refer to specific lectures, so it would be useful to have them handy before you begin to respond. 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 dialects employed in this course.
- You may not evaluate any 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. Act honorably.

Problem	Grade
1	
2	
3	
4	
5	
6	
7	
8	
9	
Total	

Thank you for taking cs173. May chance favor you.

Problem 1 [5 points]

On 2002-11-04, we saw that in the simply-typed lambda calculus, all programs are guaranteed to halt. This makes the type checker for that language a program that can determine whether or not programs halt (in this case, it always responds in the affirmative). Does this contradict the Halting Problem? (Please state clearly whether or not it does; if you claim it doesn't, explain why not, and if you claim it does, reconcile. Be brief and very clear!)

Problem 2 [15 points]

Consider the program

(+ 1 (*first* (*cons* true *empty*)))

This program has a type error.
Generate constraints for this program. Isolate the smallest set of these constraints that, solved together, identify the type error.
Feel free to label the sub-expressions above with superscripts for use when writing and solving constraints.

Problem 3 [15 points]

Languages like ML and Haskell support both let-based polymorphic type inference and type annotations on procedure and module boundaries. To model this, consider a new language construct with the following BNF description:

```
{annotfun {<id> : <type>} : <type> <expr>}
```

That is, instead of expecting inference to infer the type of the procedure, the programmer uses `annotfun` to explicitly declare its type.
Describe the constraint generation rule(s) for `annotfun` expressions. (This should not be different from the rule for unannotated procedures.)

Do these additional rules affect the unification-based constraint solver? Why or why not?

Problem 4 [10 points]

Consider the type judgments discussed in the lecture notes for 2002-11-01. These rules are for an *eager* language. Consider the *lazy* version of the language instead. Indicate which of the judgments would change, and how, by writing the modified judgments below. Do not write judgments that do not change between the two versions of the language. Pay special attention to the typing rules for

- function definition
- function application

For each one, if you believe the existing rule does not change, explain why not. (If you believe neither rule changes, you can answer both parts together.)

Problem 5 [15 points]

We have discussed why, at least on paper, mark-and-sweep is an inferior garbage collection strategy for languages that permit implementations to move data in memory (such as ones that do not expose pointer operations). Yet even for such languages, most memory managers use a mark-and-sweep strategy for their large-object space.
Give one reason why mark-and-sweep might be preferable on such objects.

State two standard objections to mark-and-sweep, and explain why they don't apply in this context. For each reason, first state the objection, then explain why it doesn't apply.

Problem 6 [10 points]

Suppose we have multiple kinds of data of different sizes. In the heap representations we studied in class, the type tag is always at the first address (i.e., at offset 0) that represents a value. Should it matter where the tag resides? For instance, why not put the tag at the last address rather than the first? (The more concrete you are, the more brief you can be.)

Problem 7 [15 points]

In this course, we studied the use of *hygienic* macro systems in Scheme. Consider the macro for **or**, presented on 2002-11-22 (where it was called **or%**). Create a fragment of code that produces different results under hygienic and "unhygienic" expanders. Present the original program, the expanded code under each expander, and the result of evaluating each expanded program.

Problem 8 [15 points]

Starting with the language *FAE*, which has only functions and addition, we add the new construct `iota`. `iota` reflects the amount of time that has elapsed since the beginning of program execution. Programmers can therefore employ `iota` to implement simple forms of profiling, pre-emptive multi-tasking, etc.
Concretely, the initial value of `iota` is `0`. Its value increases by one on every function application. References to `iota` evaluate to the current value of the counter. Thus, the program

```
{+ iota
   {{fun {dummy}
      iota}
    1729}}
```

evaluates to `1` (the reference to `iota` on the left-hand-side evaluates to `0`, while the one inside the application evaluates to `1`).
Extend the following interpreter with the `iota` construct. Clearly mark all additions and changes. Remember to update the abstract syntax. You may *not* use mutation anywhere in your interpreter!

```
(define-datatype FAE FAE?
  [num (n number?)]
  [add (lhs FAE?) (rhs FAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body FAE?)]
  [app (fun-expr FAE?) (arg-expr FAE?)])

(define-datatype Env Env?
  [mtSub]
  [aSub (name symbol?)
        (value FA-value?)
        (env Env?)])

(define-datatype FA-value FA-value?
  [numV (n number?)]
  [closureV (param symbol?)
            (body FAE?)
            (cache Env?)])

;; parse : sexp → FAE
(define (parse sexp)
  (cond
    [(symbol? sexp) (id sexp)]
    [(number? sexp) (num sexp)]
    [(list? sexp)
     (case (first sexp)
       [(+) (add (parse (second sexp))
                 (parse (third sexp)))]
       [(with) (parse `{{fun {,(first (second sexp))}
                             ,(third sexp)}
                           ,(second (second sexp))})]
       [(fun) (fun (first (second sexp)) (parse (third sexp)))]
       [else (app (parse (first sexp)) (parse (second sexp)))])]))
```

```
;; numV-n : FAE [numv] → number
(define (numV-n value)
  (cases FA-value value
    [numV (n) n]
    [else (error 'numV-n "not a number")]))

;; numV+ : numV numV → numV
(define (numV+ n1 n2)
  (numV (+ (numV-n n1) (numV-n n2))))

;; lookup : symbol Env → FA-value
(define (lookup name env)
  (cases Env env
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-env))]))

;; interp : FAE Env → FA-value
(define (interp expr env)
  (cases FAE expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (param body)
         (closureV param body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)]
                 [define arg-val (interp arg-expr env)])
           (cases FA-value fun-val
             [closureV (cl-param cl-body cl-cache) (interp cl-body
                                                           (aSub cl-param
                                                                 arg-val
                                                                 cl-cache))]
             [else (error 'interp "can only apply functions")]))]))
```

Problem 9

[2 points]

Name any two kinds of camel.