Name:

Midterm 1

CMSC 430
Introduction to Compilers
Spring 2012

March 14, 2012

Instructions

This exam contains 8 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		32
2		34
3		34
Total		100

Question 1. Short Answer (32 points).

a. (8 points) In at most 3 sentences, explain the difference between a compiler front-end and a compiler back-end.

> **Answer:** The front-end of the compiler is responsible for lexing and parsing, and the back-end of the compiler is responsible for code generation.

b. (8 points) As part of project 2, you extended Rube to include type annotations for fields and local variables. However, fields and local variables included no explicit initialization. But then, how can Rube initialize fields and local variables to ensure their initial values are type safe? Explain your answer briefly.

> **Answer:** Fields and local variables can be safely initialized with `nil`, since the type $\perp$ of `nil` is a subtype of all other types.

c. **(8 points)** Explain briefly what a *basic block* is.

Answer: A basic block is a sequence of instructions with no branches from it except from the last statement, and no branches into it except to the beginning of the block.

d. **(8 points)** In project 1, we allowed dimensional quantities to be converted between units. In project 2, we included subtyping so that subtypes could be used where supertypes were expected. Compare and contrast these two ideas. In what ways are they similar, and how are they different? Give at least one similiarity and one difference. Write at most a few sentences.

Answer: There are many possible answers.

- Both ideas allow values of different kinds to be converted to or from each other.
- In project 1, we only had explicit casts, whereas in project 2 subtyping could be used both explicitly and implicitly.
- In subtyping, one direction is always valid (sub- to supertype), and the other direction may not be valid. With unit conversion, both directions of conversion are always possible.
- Subtyping in project 2 relates many different types, whereas by design, in project 1 we only converted to or from SI base units.

Question 2. Parsing (34 points). Consider the grammar for the lambda calculus, which we can write down with the following `.mly` file (actions omitted):

```
...
%%
expr:
| ID              (* variable; production 1 *)
| LAM ID DOT expr (* function binding; production 2 *)
| expr expr       (* function application; production 3 *)
;
```

a. (4 points) List all the *tokens* that are referred to in the grammar.

Answer: `LAM`, `ID`, and `DOT`.

b. (10 points) Suppose we defined the following data type for abstract syntax trees for lambda calculus:

```
type expr =
  | Var of string
  | Lam of string * expr
  | App of expr * expr
```

Fill in the actions for the lambda calculus grammar so that the parser will generate abstract syntax trees of type `expr`.

```
...
%%
expr:

| ID {                                              }


| LAM ID DOT expr {                                 }


| expr expr {                                       }
;
```

Answer:

```
expr: ID { Var($1) } | LAM ID DOT expr { Lam($2, $4) } | expr expr { App($1, $2) }
```

c. (15 points) Following is the (slightly simplified) `.output` file produced when we run `ocamlyacc` on this grammar. We've numbered the productions above to correspond with how the productions are referred to in the output file:

state 1 %entry% : . expr (4) ID shift 3 LAM shift 4 . error expr goto 5	**state 3** expr : ID . (1) . reduce 1	**state 4** expr : LAM . ID DOT expr (2) ID shift 6 . error
state 5 expr : expr . expr (3) %entry% : expr . (4) ID shift 3 LAM shift 4 \$end reduce 4 expr goto 7	**state 6** expr : LAM ID . DOT expr (2) DOT shift 8 . error	**state 7** *shift/reduce conflict* *(shift 3, reduce 3) on ID* *(shift 4, reduce 3) on LAM* expr : expr . expr (3) expr : expr expr . (3) ID shift 3 LAM shift 4 \$end reduce 3 expr goto 7
state 8 expr : LAM ID DOT . expr (2) ID shift 3 LAM shift 4 . error expr goto 9	**state 9** *shift/reduce conflict* *(shift 3, reduce 2) on ID* *(shift 4, reduce 2) on LAM* expr : LAM ID DOT expr . (2) expr : expr . expr (3) ID shift 3 LAM shift 4 \$end reduce 2 expr goto 7	

Fill in the following table to show the steps taken to parse the input string `LAM ID DOT ID $end`. That is, on each row list (a) the state stack, with the top of the stack on the left; (b) the input with a "." indicating what input characters have been scanned and which remain to be scanned; and (c) the action that leads to the next state stack; an action should either be "shift" or "reduce n", where n is a production number. Stop filling out the table when you reach the action "reduce 4". We've filled out some of the table to get you started. (Note that there may or may not be some extra rows in the table.)

State stack	**Input**	**Action**
1	. LAM ID DOT ID $end	shift
4, 1	LAM . ID DOT ID $end	shift
6,4,1	LAM ID . DOT ID	shift
8, 6, 4, 1	LAM ID DOT . ID $end	shift
3, 8, 6, 4, 1	LAM ID DOT ID . $end	reduce 1
9, 8, 6, 4, 1	LAM ID DOT ID . $end	reduce 2
5, 1	LAM ID DOT ID . $end	reduce 4

d. (5 points) The parser generated from this grammar has shift/reduce conflicts in state 7. What specific problem with the grammar is exhibited by these conflicts? The parser generator has resolved the conflict by shifting on *ID* or *LAM*; will this resolution lead to correct parsing, according to the conventions of lambda calculus? Explain.

> **Answer:** The conflicts in state 7 are due to ambiguity in the grammar, because we have not specified the associativity of function application. In this case, shifting is incorrect, as it corresponds to right associativity, whereas in lambda calculus, function application should be left-associative.

Question 3. Operational semantics and type checking (34 points). Consider extending the simply typed lambda calculus to include lists and a corresponding type:

$$\begin{array}{rcl} e & ::= & n \mid x \mid \lambda x.e \mid e\ e \mid [] \mid e :: e \mid \mathit{hd}\, e \mid \mathit{tl}\, e \\ t & ::= & \mathit{int} \mid t\ \mathit{list} \mid t \to t \\ A & ::= & \cdot \mid x : t, A \end{array}$$

Here [] is the empty list; $e_1 :: e_2$ creates a list with head e_1 and tail e_2; the expression $\mathit{hd}\, e$ returns the head of list e; and the expression $\mathit{tl}\, e$ returns the tail of list e. The type $t\ \mathit{list}$ is the type of a list whose elements have type t.

Here are the type rules for this language:

$$\textsc{Int}\ \frac{}{A \vdash n : \mathit{int}} \qquad \textsc{Var}\ \frac{x \in \mathit{dom}(A)}{A \vdash x : A(x)} \qquad \textsc{Lam}\ \frac{x : t, A \vdash e : t'}{A \vdash \lambda x.e : t \to t'} \qquad \textsc{App}\ \frac{A \vdash e_1 : t \to t' \qquad A \vdash e_2 : t}{A \vdash e_1\ e_2 : t'}$$

$$\textsc{Nil}\ \frac{}{A \vdash [] : t\ \mathit{list}} \qquad \textsc{List}\ \frac{A \vdash e_1 : t \qquad A \vdash e_2 : t\ \mathit{list}}{A \vdash e_1 :: e_2 : t\ \mathit{list}} \qquad \textsc{Hd}\ \frac{A \vdash e : t\ \mathit{list}}{A \vdash \mathit{hd}\, e : t} \qquad \textsc{Tl}\ \frac{A \vdash e : t\ \mathit{list}}{A \vdash \mathit{tl}\, e : t\ \mathit{list}}$$

a. (10 points) Write down a derivation that shows $\cdot \vdash (\lambda x.\lambda y.x :: \mathit{tl}\, y) : \mathit{int} \to \mathit{int\ list} \to \mathit{int\ list}$.

Answer:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : \mathit{int\ list}, x : \mathit{int} \vdash x : \mathit{int}} \qquad \dfrac{\dfrac{}{y : \mathit{int\ list}, x : \mathit{int} \vdash y : \mathit{int\ list}}}{y : \mathit{int\ list}, x : \mathit{int} \vdash \mathit{tl}\, y : \mathit{int\ list}}}{y : \mathit{int\ list}, x : \mathit{int} \vdash (x :: \mathit{tl}\, y) : \mathit{int\ list}}}{x : \mathit{int} \vdash (\lambda y.x :: \mathit{tl}\, y) : \mathit{int\ list} \to \mathit{int\ list}}}{\cdot \vdash (\lambda x.\lambda y.x :: \mathit{tl}\, y) : \mathit{int} \to \mathit{int\ list} \to \mathit{int\ list}}$$

b. (4 points) Consider the following operational semantics rule:

$$\frac{}{\mathit{hd}\, (e_1 :: e_2) \to e_2}$$

Show that this rule is incorrect by showing that using it will cause Preservation to be violated, i.e., give an expression e such that $A \vdash e : t$ and $e \to e'$ but $A \nvdash e' : t$. Explain your answer briefly, and explain how to fix the above rule.

Answer: Consider $\mathit{hd}\, (1 :: [])$. This expression has type *int*, but if we evaluate it one step using the above rule, the result is [], which has type *int list*. The rule should have e_1 on the right-hand side of the arrow.

c. (10 points) Suppose we were to add floating point numbers to our language, with appropriate subtyping rules:

$$\begin{array}{lcl} e & ::= & f \mid n \mid x \mid \lambda x.e \mid e\ e \mid [] \mid e :: e \mid hd\, e \mid tl\, e \\ t & ::= & \mathit{float} \mid \mathit{int} \mid t\ \mathit{list} \mid t \to t \end{array}$$

$$\frac{}{t \leq t}\ \textsc{Refl} \qquad \frac{}{\mathit{int} \leq \mathit{float}}\ \textsc{Sub-base} \qquad \frac{t_2 \leq t_1 \qquad t_1' \leq t_2'}{t_1 \to t_1' \leq t_2 \to t_2'}\ \textsc{Sub-fun} \qquad \frac{t_1 \leq t_2}{t_1\ \mathit{list} \leq t_2\ \mathit{list}}\ \textsc{Sub-list}$$

For each of the following subtyping relationships, write "yes" or "no" to indicate whether or not the relationship holds according to the rules above.

$\mathit{int} \leq \mathit{int}$	*yes*
$\mathit{float} \leq \mathit{int}$	*no*
$\mathit{int\ list\ list} \leq \mathit{float\ list\ list}$	*yes*
$\mathit{int} \to \mathit{float} \leq \mathit{float} \to \mathit{float}$	*no*
$\mathit{int} \to \mathit{int} \to \mathit{int} \leq \mathit{float} \to \mathit{int} \to \mathit{float}$	*no*

d. (10 points) Write down an OCaml data type `typ` and a function `is_subtype t1 t2 : typ -> typ -> bool` that returns true iff and only `t1` is a subtype of `t2` according to the rules above.

Answer:

```
type typ = TFloat | TInt | TList of typ | TFun of typ * typ

let rec is_subtype t1 t2 = match t1, t2 with
| TFloat, TFloat | TInt, TInt | TInt, TFloat -> true
| TList t1, TList t2 -> is_subtype t1 t2
| TFun (t1, t1'), TFun(t2, t2') ->
  (is_subtype t2 t1) && (is_subtype t1' t2')
| _ -> false
```