Instructions

This exam contains 10 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.

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
Question 1. Short Answer (25 points).

a. (5 points) Briefly explain what a virtual method table (or vtable) is and what it’s used for.

Answer:

It is a collection of methods for a particular class. Each instance of a class has a pointer to the virtual method table for that class. When one of an object’s instance methods is invoked, it is resolved by looking it up in the virtual method table.

b. (5 points) List three potential goals of optimization in a compiler.

Answer:

Improve running time, decrease memory usage, reduce code size. Other reasonable answers were also acceptable.
c. (5 points) Briefly explain what the progress theorem is.

d. (5 points) What do mutation and crossover have to do with automated program repair, as discussed in class? Explain very briefly.
e. (5 points) Briefly explain what an *activation record* is and list 3 items in an activation record.
Question 2. Type Systems (25 points).

a. (8 points) Assume that \texttt{int} < \texttt{float}. Write down every type \( t \) such that \( t \leq \texttt{int} \rightarrow \texttt{float} \rightarrow \texttt{float} \), following standard subtyping rules.

Answer:

\[
\begin{align*}
\text{int} \rightarrow \text{float} \rightarrow \text{int} \\
\text{int} \rightarrow \text{float} \rightarrow \text{float} \\
\text{float} \rightarrow \text{float} \rightarrow \text{int} \\
\text{float} \rightarrow \text{float} \rightarrow \text{float}
\end{align*}
\]

b. (2 points) Assume that \texttt{int} < \texttt{float}. Write down every type \( t \) such that \( t \leq \texttt{int} \texttt{ref} \rightarrow \texttt{float} \texttt{ref} \), following standard subtyping rules.

Answer:

\[
\begin{align*}
\text{int} \texttt{ref} \rightarrow \texttt{float} \texttt{ref}
\end{align*}
\]
c. **(10 points)** Fill in the following table with either an *untyped* (i.e., no type parameter annotations) lambda calculus term (on the left) or its corresponding type according to the type inference algorithm we saw in class (on the right).

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
</tr>
<tr>
<td>( \lambda x.x )</td>
<td>( \alpha \rightarrow \beta \rightarrow \beta )</td>
</tr>
<tr>
<td>( \lambda x.\lambda y.y \ x )</td>
<td></td>
</tr>
<tr>
<td>( \lambda x.x \ 3 )</td>
<td></td>
</tr>
</tbody>
</table>

d. **(5 points)** Recall the simply typed lambda calculus:

\[
\begin{align*}
e & ::= n \mid x \mid \lambda x : t.e \mid e\ e \\
t & ::= int \mid t \rightarrow t \\
A & ::= \emptyset \mid x : t, A
\end{align*}
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Context</th>
<th>Premises</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td></td>
<td>( A \vdash n : \text{int} )</td>
</tr>
<tr>
<td>VAR</td>
<td></td>
<td>( A \vdash x : A(x) )</td>
</tr>
<tr>
<td>LAM</td>
<td>( x : t, A \vdash e : t' ) \quad ( A \vdash (\lambda x : t.e) : t' )</td>
<td>( A \vdash e_1 : t' \rightarrow t' \quad A \vdash e_2 : t )</td>
</tr>
<tr>
<td>APP</td>
<td>( A \vdash e_1 : t' \rightarrow t' \quad A \vdash e_2 : t )</td>
<td>( A \vdash e_1 \ e_2 : t' )</td>
</tr>
</tbody>
</table>

Draw a derivation that the following type judgment holds, where \( A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \). (You can draw the derivation upward from the judgment, and you can write \( i \) instead of \( \text{int} \) to save time):

\[ A \vdash (\lambda x : \text{int}. \ x) \ 1 : \text{int} \rightarrow \text{int} \]
Question 3. Interpreter Implementation (10 points). Below is a snippet of the bytecode interpreter code from 06-codegen-2.ml.

Suppose we make the unfortunate decision to modify our bytecode language to have a special undefined value, like JavaScript. We begin by introducing a new type, intOrUndef, to stand for either the undefined value or an integer; we add a new instruction, IIfUndef (r, n); and we adjust the types src, heap, and regs appropriately:

The desired semantics is as follows:

- **IIfUndef (r, n)** branches by n if r contains Undef, otherwise it falls through.
- If Undef if used as either argument to addition, the result should be Undef.
- If Undef is used as the guard of IIfZero, it should be treated as false (i.e., as a non-zero value).

Rewrite the cases in run_instr for IIfUndef, IAdd, and IIfZero to implement this semantics. You can write a helper function if you want. You do not need to implement any other parts of run_instr.

let rec run_instr (h:heap) (rs:regs) = function

Answer:

```ocaml
let stupid_add = function
| Int x, Int y -> x + y
| _ -> Undef
```

```ocaml
let rec run_instr instr h rs = match instr with
| IAdd r1, r2, r3 -> Hashtbl.replace rs r1 (stupid_add (Hashtbl.find rs r2), (Hashtbl.find rs r3)); None
| IIfZero r, n -> if (Hashtbl.find rs r) = 0 then Some n else None
| IIfUndef r, n ->
  match (Hashtbl.find rs r) with
  | Undef -> Some n
  | _ -> None
| ...

let rec run instr (h:heap) (rs:regs) = function
```

```ocaml
| ILoad of src -> (dst, src)
| IStore of dst -> dst
| IAdd of reg * reg * reg -> dst, src1, src2
| IMul of reg * reg * reg -> dst, src1, src2
| IIfZero of reg * int -> guard, target
| IJump of int -> target
| IMove of reg * reg -> dst, src
```
Question 4. Code Generation (25 points). Below is more code from `06-codegen-2.ml`, showing the input expression and part of the compiler.

```ocaml
type expr =
  | EInt of int
  | EPlus of expr * expr
  | EMul of expr * expr
  | EId of string
  | EAssn of string * expr
  | ESeq of expr * expr
  | EIfZero of expr * expr * expr

type symtbl = (string * int) list

let rec comp expr (st:symtbl) =
  function
  | EInt n →
    let r = next_reg () in
    (r, [ILoad ('Reg r, 'Const n)]
  | EPlus (e1, e2) →
    let (r1, p1) = comp expr st e1 in
    let (r2, p2) = comp expr st e2 in
    let r = next_reg () in
    (r, p1 @ p2 @ [IAdd ('Reg r, 'Reg r1, 'Reg r2)])
  | EIfZero (e1, e2, e3) →
    let (r1, p1) = comp expr st e1 in
    let (r2, p2) = comp expr st e2 in
    let (r3, p3) = comp expr st e3 in
    let r = next_reg () in
    (r, p1 @ [IIfZero ('Reg r1, (2 + (List.length p3))) @
               IMov ('Reg r, 'Reg r3); IJmp (1 + (List.length p2))] @
       p2 @ [IMov ('Reg r, 'Reg r2)])
```

a. (10 points) Suppose we extend the source language with a `repeat-until loop` `ERepeat(e1, e2)`, meaning “repeat `e1` until `e2` becomes non-zero.” Note that a repeat-until loop always executes the body `e1` at least once (so it evaluates `e1`; checks if `e2` is non-zero; if not evaluates `e1` again; etc). Write a case of `comp_expr` that compiles `ERepeat`. The loop itself should evaluate to 0.

```ocaml
let rec comp_expr (st:symtbl) =
  function
  | ERepeat (e1, e2) →
    ...
```
b. (15 points) Now consider again adding an undefined value to the language:

```ml
type expr =
| EUndef
| ...
```

Write the EUndef case of `comp_expr`. Also, rewrite the EPlus case of `comp_expr` to implement the Undef semantics without relying on the special IAdd handling that understands Undef. That is, your compiled output code should only call IAdd with integer arguments.

```ml
let rec comp_expr (st:symtbl) = function
...
| EUndef →
```

```ml
... | EPlus (e1, e2) →
```
**Question 5. Optimization (15 points).** In each row of the table, perform the indicated optimization (and only that optimization), writing down the optimized code on the right-hand side of the table. To reduce writing, we write \( r_n \) instead of ‘Reg \( n \), and we write \( n \) instead of ‘Const \( n \).

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Optimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILoad ((r_0, 42))</td>
<td>Copy propagation</td>
</tr>
<tr>
<td>IMov ((r_1, r_0))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_2, r_0, r_1))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_3, r_1, r_2))</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>IMul ((r_4, r_1, r_2))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_5, r_1, r_2))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ILoad ((r_0, 42))</td>
<td>Constant folding</td>
</tr>
<tr>
<td>ILoad ((r_1, 3))</td>
<td></td>
</tr>
<tr>
<td>ILoad ((r_2, r_0, r_1))</td>
<td></td>
</tr>
<tr>
<td>IMul ((r_4, r_1, r_2))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_5, r_1, r_3))</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ILoad ((r_0, 0))</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>ILoad ((r_1, 1))</td>
<td></td>
</tr>
<tr>
<td>ILoad ((r_2, 2))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_3, r_0, r_1))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_4, r_0, r_2))</td>
<td></td>
</tr>
<tr>
<td>IAdd ((r_5, r_1, r_3))</td>
<td></td>
</tr>
<tr>
<td>(* assume only r_5 is live *)</td>
<td></td>
</tr>
</tbody>
</table>