Midterm II

- Please read all instructions (including these) carefully.
- There are six questions on the exam, each worth between 15 or 20 points. You have 3 hours to work on the exam.
- The exam is closed book, but you may refer to your four sheets of prepared notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. There are no "tricky" problems on the exam—each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary.

NAME: Sample solution

SID or SS#: ______________________

Problem	Max points	Points
1	20	
2	20	
3	20	
4	20	
5	20	
6	15	
TOTAL	115	

1. **Overloading Resolution** (20 points)
This problem considers the design of an overloading mechanism for a small, C-like language. The grammar of the language is:

$$\begin{aligned}
\texttt{Program} &\rightarrow \texttt{Def; Program} \mid \epsilon \\
\texttt{Def} &\rightarrow \texttt{funid(id : type, \ldots, id : type) : type = Expr} \\
\texttt{Expr} &\rightarrow \texttt{integer} \mid \texttt{id} \mid \texttt{funid}(\mathrm{E}_1, \ldots, \mathrm{E}_n) \mid \mathrm{E} + \mathrm{E} \mid \&\mathrm{E} \mid *\mathrm{E}
\end{aligned}$$

There are two disjoint classes of identifiers: function identifiers (funid's) and data identifiers (id's). The scope of a function identifier is from the point of definition to the end of the program. Data identifiers are local to a function definition. The types of the language are:

$$\texttt{T} \rightarrow \texttt{Int} \mid \texttt{Pointer(T)}$$

The operation &`e` takes the address of `e` and ∗`e` dereferences the pointer `e`. The + operator adds a pair of integers. A function `f` may have many different definitions, but the number and/or types of the arguments for each definition must be distinct. For example, there can be functions `f(int,int) : int` and `f(int, pointer(int)): int`, but there cannot be two functions `f` with argument types `f(int,int)`.

Fill in the skeleton algorithm for typechecking and overloading resolution given on the next page. Add arguments to the functions if necessary; you should decide and clearly state what, if anything, the functions return. Use any clear programming notation. You may assume you have a symbol table implementation with reasonable operations. Focus on computing the types—don't worry about what to do if identifiers are undeclared, or type checking fails, or any other error condition. In a case branch, you may refer to the symbols on the left-hand side of the `=>`. For example,

```
e1 + e2 => ... your computation using e1 and e2 ...
```

```
main(e) =  /* main is called with the program to typecheck */
{
   fnenv = new FnEnvironment;
   tc_program(e, fnenv)
}  /* Note: The error checking is not required for full credit*/

tc_program(e, fnenv)  =
{
    case e of
       D; P    => tc_def(D, fnenv); tc_program(P, fnenv);

       epsilon => /* done */
}

tc_def(e, fnenv) =
{
    case e of
         funid(id1 : type1,...,idn : typen) : restype = expr  =>
           fnenv->add(funid, (type1, ..., typen), restype);
           idenv = new IdEnvironment;
           for i = 1 .. n: idenv->add(idi, typei);
           bodytype = tc_expr(expr, idenv, fnenv);
           if bodytype != restype then error("bad type");
}

type tc_expr(e, idenv, fnenv) =
{
    case e of
         integer => return int_type;

         id => return idenv->lookup(id);

         funid(e1,...,en) =>
           for i = 1 .. n: etypei = tc_expr(ei, idenv, fnenv);

           // get list of definitions for funid
           allfns = fnenv->lookup(funid);
           foreach fndef in allfns:
             if fndef.arguments = (etype1, ..., etypen) then
               return fndef.restype;
           error("No matching function");

         e1 + e2 =>
           atype1 = tc_expr(e1, idenv, fnenv);
           atype2 = tc_expr(e2, idenv, fnenv);
           if atype1 != int_type || atype2 != int_type then
             error("Bad add");
```

```
              return int_type;

            &e1 =>
              ptype = tc_expr(e1, idenv, fnenv);
              return pointer(ptype);

            *e1 =>
              stype = tc_expr(e1, idenv, fnenv);
              if stype = pointer(x) then return x
              else error("Bad pointer dereference");

  }
```

2. **Attribute Grammars** (20 points) The following grammar defines a small functional language

   `E → (fun id : type E) | (E E) | int | id`

   The notation `(fun id: type E)` defines a function where argument `id` is declared to have type `type`. The types for this language are:

   `Type ::= Int | Type → Type`

   The type rules are:

$$\frac{A(x) = t}{A \vdash x : t} \qquad \frac{i \text{ is an integer}}{A \vdash i : Int} \qquad \frac{A[x \leftarrow t_1] \vdash e : t_2}{A \vdash (fun\ x : t_1\ e) : t_1 \rightarrow t_2} \qquad \frac{\begin{array}{l} A \vdash e_1 : t_1 \rightarrow t_2 \\ A \vdash e_2 : t_1 \end{array}}{A \vdash (e_1\ e_2) : t_2}$$

   Give an attribute grammar that assigns the type of an expression `e` to the attribute `e.type`. You may assume that every variable in the expression is introduced by an enclosing `fun x ...`. If there is a type error in `e`, then assign `e.type` the value `wrong`. You may use any reasonable and clear pseudo-code notation for the rules.

```
E0 -> (fun id : type E1)
      { E1.env = E0.env U { (id, type) };
        E0.type =
          if E1.type == wrong then wrong
          else type -> E1.type;
      }

E0 -> (E1 E2)
      { E1.env = E0.env; E2.env = E0.env;
        E0.type =
          if E1.type == E2.type -> x then x
          else wrong;
      }

E0 -> int { E0.type = int; }

E0 -> id { E0.type = t such that (id, t) is in E0.env, or wrong
                     if no such pair in E0.env
         }
```

3. **Polymorphism** (20 points)

(a) This part uses the same language as the previous question, except the type declarations are omitted. Give principal types for each of the following functions:

```
(fun z (fun y z(y)))
```
$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

```
(fun x (fun y ((x y) y)))
```
$(\alpha \rightarrow \alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

```
(fun z (fun x (x (fun y (y z)))))
```
$\alpha \rightarrow (((\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \gamma) \rightarrow \gamma$

(b) Give a most general unifier for the following system of type equations:

$$\begin{aligned} \pi &= \alpha \\ \alpha &= \beta \rightarrow (\gamma \rightarrow \gamma) \\ (\sigma \rightarrow \sigma) \rightarrow \sigma &= \pi \end{aligned}$$

$$\begin{aligned} \sigma &\Rightarrow \gamma \rightarrow \gamma \\ \beta &\Rightarrow (\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma) \\ \pi &\Rightarrow ((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)) \rightarrow (\gamma \rightarrow \gamma) \\ \alpha &\Rightarrow ((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)) \rightarrow (\gamma \rightarrow \gamma) \end{aligned}$$

4. **Cool Type Checking** (20 points)

Typecheck the following simple Cool fragments, giving the proof trees. Show the structure of the proof and the types of all subexpressions; we don't expect you to show (or remember!) exactly all the side conditions of the rules. Any nodes of the tree that are type errors should be clearly indicated. You should assume that the type of any such nodes is `Object` (you may thus get cascading errors).

(a) Recall `copy` is a method of the `Object` class; `copy` has signature `copy(): SELF_TYPE`.

```
let x : Int <- 3, y : Int in
    x.copy() + y
end
```

$$\dfrac{\dfrac{}{\vdash 3 : Int} \quad \dfrac{\dfrac{\dfrac{}{[x : Int, y : Int] \vdash \texttt{x : Int}}}{[x : Int, y : Int] \vdash \texttt{x.copy() : Int}} \quad \dfrac{}{[x : Int, y : Int] \vdash \texttt{y : Int}}}{[x : Int, y : Int] \vdash \texttt{x.copy() + y : Int}}}{\vdash \texttt{let x: Int <- 3, y : Int in x.copy() + y end : Int}}$$

(b) Recall `concat` is a method of the `String` class; `concat` has signature `concat(String) : String`. For this question, show only proof trees for features in class `B`.

```
class A is
  a1 : Int;
  m1(x : String): String is
    x.concat("fun")
  end;
end;

class B inherits A is
  a2 : A <- new SELF_TYPE;
  m1(y : String): String is
    self@A.m1("this ".concat(y))
  end;
end;
```

(Unimportant environment information is omitted)

$$\dfrac{\dfrac{}{[a1 : Int, a2 : A] \vdash \texttt{new SELF\_TYPE : SELF\_TYPE}_B}}{[a1 : Int, a2 : A] \vdash \texttt{a2 : A <- new SELF\_TYPE ;}}$$

$$\dfrac{\dfrac{\dfrac{}{[y : String] \vdash \texttt{self : SELF\_TYPE}_B} \quad \dfrac{\dfrac{}{[y : String] \vdash \texttt{"this " : String}} \quad \dfrac{}{[y : String] \vdash \texttt{y : String}}}{[y : String] \vdash \texttt{"this ".concat(y) : String}}}{[y : String] \vdash \texttt{selfA.m1("this ".concat(y)) : String}}}{\vdash \texttt{m1(y : String) : String is ...end ;}}$$

(c) Show proof trees for all features of `A`.

```
class A is
  a1 : Int <- "fun";
  m1(x : A): SELF_TYPE is
    begin
      x.m1(x+1);
      x;
    end
  end;
end;
```

(Unimportant environment information is omitted)

$$\dfrac{\dfrac{}{\vdash \text{"}fun\text{"} : String}}{[a1 : Int] \vdash \texttt{a1 : Int <- "fun"} \ ; ERROR}$$

$$\dfrac{\dfrac{\dfrac{}{[x : A] \vdash \texttt{x : A}} \quad \dfrac{\dfrac{}{[x : A] \vdash \texttt{x : A}} \quad \dfrac{}{[x : A] \vdash \texttt{1 : Int}}}{[x : A] \vdash \texttt{x + 1 : Object}ERROR}}{[x : A] \vdash \texttt{x.m1(x+1) : Object}ERROR} \quad \dfrac{}{[x : A] \vdash \texttt{x : A}}}{\dfrac{[x : A] \vdash \texttt{begin x.m1(x+1); x; end : A}}{\vdash \texttt{m1(x : A): SELF\_TYPE is ... end} \ ; ERROR}}$$

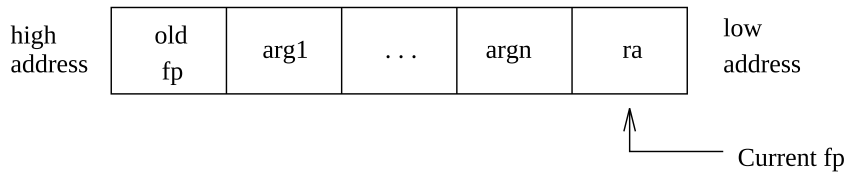
5. **Code Generation** (20 points)

In class we described how to generate stack machine code for the following language:

$$\begin{array}{rcl} \texttt{P} & \rightarrow & \texttt{D;P} \mid \texttt{D} \\ \texttt{D} & \rightarrow & \texttt{def id(ARGS)} = \texttt{E;} \\ \texttt{ARGS} & \rightarrow & \texttt{id, ARGS} \mid \texttt{id} \\ \texttt{E} & \rightarrow & \texttt{int} \mid \texttt{id} \mid \texttt{if } \texttt{E}_1 = \texttt{E}_2 \texttt{ then } \texttt{E}_3 \texttt{ else } \texttt{E}_4 \mid \\ & & \texttt{E}_1 + \texttt{E}_2 \mid \texttt{E}_1 - \texttt{E}_2 \mid \texttt{id}(\texttt{E}_1, \ldots, \texttt{E}_\texttt{n}) \end{array}$$

You may assume that within a function body, the arguments to the function are named *arg1*, *arg2*, etc., with the index giving the position in the function's argument list. You may also assume that functions are prefixed by *fun*; e.g., *funfib*, *funfac*, etc. Thus, it is possible to distinguish integer arguments and function names syntactically.

The activation record for a function invocation `funX(arg1, ..., argn)` has the form shown below. Every activation record entry is 4 bytes long. Note that the order of arguments in the AR is different from that given in lecture.



Use the following subset of MIPS assembly in answering this question. Register names are prefixed with a "$". Use $fp for the frame pointer, $sp for the stack pointer, and $a0 for the accumulator. You may use any register names you wish for temporary values.

instruction	*meaning*
addiu $r1 $r2 imm	$r1 := $r2 + imm
lw $r1 offset($r2)	$r1 := load from location $r2 + offset
sw $r1 offset($r2)	store $r1 at location $r2 + offset
label:	"label" refers to address of next instruction
la $r1 label	load the address "label" into $r1
jal label	jump and link to address "label"
jalr $r1	jump and link to address in $r1
.word imm	reserves a word of storage; initialized to integer "imm"

(a) Suppose this language is extended with global variables by adding a production `D` $\rightarrow$ `def id`. Any reference to an identifier that does not appear in the enclosing function's argument list is assumed to be global. All global variables must be declared by the program. Global names begin with the prefix *global*. All globals are integers and are initialized to 0. Define a code generation function for the productions `D` $\rightarrow$ `def id` and `E` $\rightarrow$ `id`.

```
D -> def id
  emit "{id}:  .word 0"
```

```
E -> id
  if id = "argi":
    offset = 4 * (n + 1 - i)
    emit "lw $a0 {offset}($fp)"
  else if id = "globalxxx":
    emit "la $a0 {id}"
    emit "lw $a0 0($a0)"
```

```
Note: in the strings above, {x} gets replaced by the value of variable x.
```

(b) Now suppose, in addition, first class functions are added to the language. The function call syntax becomes $\mathtt{E} \rightarrow \mathtt{E_0(E_1, \ldots, E_n)}$, so that the function to be called is a computed value. The semantics for function call becomes

- Evaluate $\mathtt{E_0, E_1, \ldots, E_n}$ in that order, giving values $v_0, v_1, \ldots, v_n$.
- Compute $v_0(v_1, \ldots, v_n)$.

In an expression, the use of an identifier naming a function loads the address of the function into the accumulator. Define a code generation function for the productions `E` $\rightarrow$ `id` and $\mathtt{E} \rightarrow \mathtt{E_0(E_1, \ldots, E_n)}$. Assume that the called function removes the activation record from the stack and that the stack pointer always points to the first unused word.

```
E -> id
  if id = "argi":
    offset = 4 * (n + 1 - i)
    emit "lw $a0 {offset}($fp)"
  else if id = "funxxx":
    emit "la $a0 {id}"
  else if id = "globalxxx":
    emit "la $a0 {id}"
    emit "lw $a0 0($a0)"
```

```
E -> E0(E1, ..., En)
  gen code for E0
  emit "sw $a0 0($sp)"
  emit "sw $fp -4($sp)"
  emit "addiu $sp $sp -8"
  for i = 1..n:
    gen code for Ei
    emit "sw $a0 0($sp)"
    emit "addiu $sp $sp -4"
  offset = (n + 2) * 4
  emit "lw $a0 {offset}($sp)"
  emit "jalr $a0"
  emit "addiu $sp $sp 4"
```

6. **Runtime Organization** (15 points)

Consider a C-like language with functions, integers, and vectors of integers. Vectors are declared with constant sizes (e.g., `int A[100]`). Vector values are implemented as pointers to a block of memory containing the vector elements. When vectors are passed as arguments or returned as the result of functions, it is the pointer that is passed or returned. Thus, vectors are always passed by reference. Integers are always passed by value. Assume that values are communicated between functions only via function arguments and results (i.e., the language has no global variables).

For each of the following combinations of language features, state whether each of activation records, vectors, and integers should be allocated in a global static area, on the stack, or in the heap. Choose the best alternative—the one that is both correct and gives the fastest code. Give a *brief* (1 sentence) justification for your answer to each part.

```
A few points first:
The ‘fastest’ storage is the global static area, as you don’t have the
overhead of allocating it. References are via absolute, known addresses.

The stack is next fastest (allocation and freeing are simple additions).
References are via an offset from a pointer.

Heap storage is the slowest, as allocation and freeing are more expensive,
and reference is via a pointer (may need an extra load to fetch pointer).
```

(a) A language with recursive functions, where integers and vectors can be passed as arguments and returned as the results of functions.

```
AR: stack. Cannot be static because of recursion.
Ints: stack. Stored in AR.
Vectors: heap. Cannot be on stack because they outlive the called function
when they are returned (AR of called function is freed when it returns).
```

(b) A language without recursive functions, where integers and vectors can be passed as arguments and returned as the results of functions.

```
AR: static. A given function is never active more than once at the same
time (otherwise it would be recursive). So we can allocate one AR only
per function, in the static area.
Ints: static. In the AR as usual.
Vectors: heap. They outlive the called function. They cannot be stored in
the static area because a given function may be called twice and
should return a different vector each time (those who gave this answer,
with the correct justification, got an extra point, we considered
that the ‘static’ answer was correct too).
```

(c) A language with recursive functions, where integers and vectors can be passed as arguments but only integers can be returned as the results of functions.

```
AR: stack. Cannot be static because of recursion.
Ints: stack. Stored in AR.
Vectors: stack. Not returned from functions, so cannot outlive them, so
can be allocated on stack. Cannot be static because several recursive calls
to a function need separate vectors.
```