CS 352 – Compilers: Principles and Practice
Final Examination, 5/4/00

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (your grade will be the percentage of your answers that are correct).

This exam is **closed book, closed notes**. You may *not* refer to any book or other materials.

You have **two hours** to complete all six (6) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature

1. (Compiler phases; 15%) Consider the following modifications to the Tiger programming language. For each proposed modification, identify the component(s) of the compiler that will need to be changed — i.e., scanner(lexical analysis), parser (syntax analysis), semantic analysis, translation to intermediate code, instruction selection, liveness analysis, register allocation, assembler/linker. *Explain* your answer!

   (a) Allow C++/Java-comment syntax, i.e., a "//" followed by text, until the end-of-line.
   Answer:

      lexer

   (b) For a let statement, require that all variable declarations precede function declarations.
   Answer:

      parser

   (c) Addition of the primitive type *float*.
   Answer:

      lexer, parser, semantics, translation, instruction selection, liveness/register allocation

   (d) Support for the SPARC architecture.
   Answer:

      instruction selection, register allocation

   (e) Addition of a repeat-until construct.
   Answer:

      lexer, parser, semantics, translation

   (f) Requiring an explicit declaration of the for-loop index variable.
   Answer:

      semantics

   (g) Addition of a switch statement (as found in C, Java, etc.).
   Answer:

      lexer, parser, semantics, translation, instruction selection

   (h) Support for separate compilation of functions.
   Answer:

      lexer, parser, semantics, assembler/linker

2. (Parsing; 30%) Consider the following simple grammar and the language it describes:

$$S \rightarrow \varepsilon \mid a \mid (S) \mid (S;\, S)$$

(a) (5%) Is this *grammar* LL(1)? Explain. [There is a simple argument.]

Answer:

No, the grammar is not LL(1) by inspection since the last two rules have a common prefix.

(b) (5%) Is this *language* LL(1)? If so, exhibit a simple LL(1) grammar for the same language; if not, explain why not.

Answer:

Yes, the language is LL(1); by factoring the common prefix we get:

$$\begin{aligned} S &\rightarrow \varepsilon \mid a \mid (ST \\ T &\rightarrow\ ) \mid;\, S) \end{aligned}$$

(c) (5%) Is this *grammar* LR(0)? Explain. [Again, there is a simple argument.]

Answer:

No, the grammar is not LR(0) because there is a shift-reduce conflict between the first rule and the other three.

(d) (5%) Is this *grammar* LR(1)? Explain. [You need not build the LR(1) state machine; rather, you can argue from follow sets.]

Answer:

Yes, the grammar is LR(1). Follow(S) is $\{\$, ;\ ,)\}$; any LR(1) lookahead set for an S rule must be a subset of Follow(S), and a and $($ are not in Follow(S). Thus, the LR(0) shift-reduce conflict is eliminated. Since S is the only non-terminal, and each of the non-ε rules results in a reduce state containing a single configuration, the grammar is LR(1).

(e) (5%) Is this *language* LR(0)? Why or why not?

Answer:

Yes, the language is LR(0), because if a language is LR(k) for any k, it is also LR(0) (though the LR(0) grammar may not be intuitive or easy to construct).

(f) (5%) Is this *language* regular? Why or why not?

Answer:

No, the language is not regular because it involves bracketing, such as $b^n d^n$, known not to be regular because it requires an unbounded number of states and so cannot be recognized by a finite automaton.

3. (Runtime management; 20%) Consider the Tiger version of *quicksort* given below, which reads a sequence of integers from standard input into an array (via the omitted function readarray), sorts the array, and prints the result (via the omitted function printarray). You can assume that function length returns the length of an array.

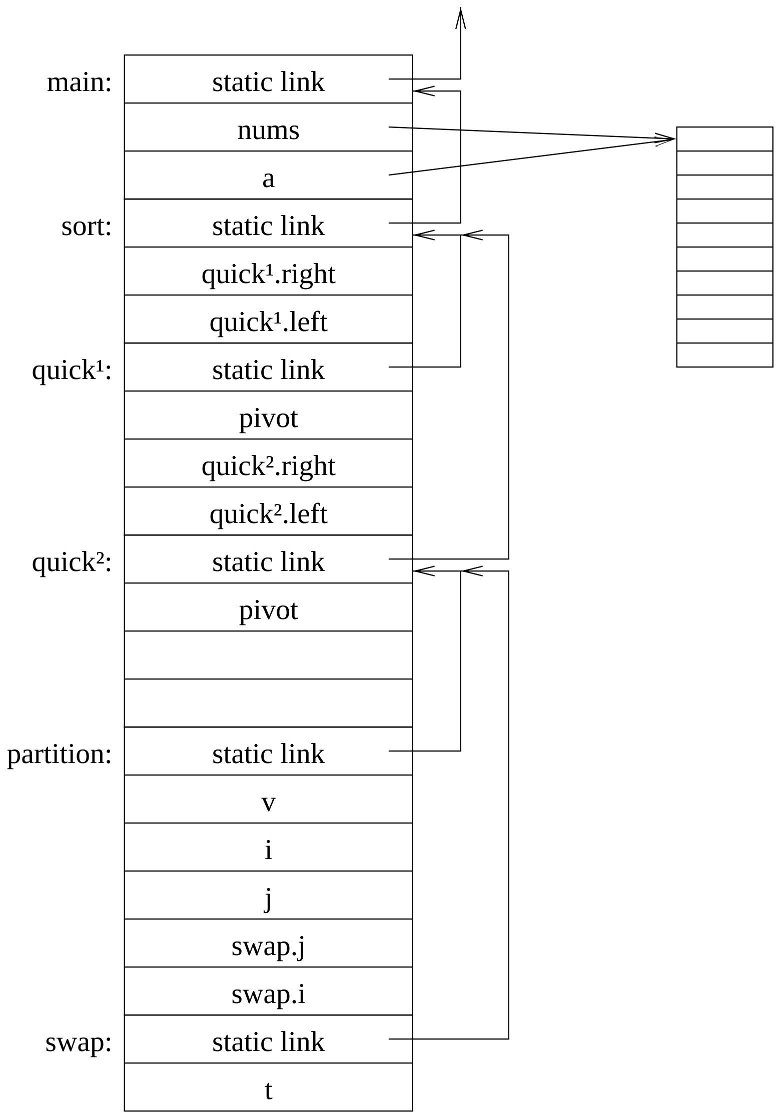
```
let type intarray = array of int
    function sort (a: intarray) =
        let function quick (left:int, right: int) =
                let function swap (i: int, j: int) =
                        let var t := a[i] in
                            a[i] := a[j]; a[j] := t
                        end
                    function partition (): int =
                        let var v := a[(left+right) / 2]
                            var i := left
                            var j := right
                        in
                            while 1 do (
                                while a[i] < v do i := i+1;
                                while a[j] > v do j := j-1;
                                if i > j then break;
                                swap(i, j); i := i+1; j := j-1
                            );
                            i
                        end
                in
                    if left < right then
                        let var pivot := partition() in
                            quick(left, pivot-1);
                            quick(pivot, right)
                        end
                end
        in
            quick(0, length(a)-1)
        end
    var nums: intarray := readarray()
in
    sort(nums);
    printarray(nums)
end
```

Show a diagram of MIPS stack frames and the variables within them at the point where *all* of the following conditions are simultaneously true:

- `sort` is active, and has called `quick`
- `quick` has called `partition`, which has returned; then `quick` has called `quick`
- `quick` has called partition
- `partition` has called swap
- `swap` is executing

Indicate where all the variables are (assume all variables are stored in memory, not in registers); but don't bother showing the values of integer variables. Show clearly (and label) where the static (lexical scoping) links are.

Answer:



4. (Translation to intermediate code; 15%) Suppose a certain compiler translates all expressions and subexpressions into intermediate code expression trees (and does not use the Nx and Cx constructors to represent expressions in different ways). Draw a picture of the IR tree that results from each of the following expressions. Assume all variables are nonescaping unless specified otherwise (e.g., the expression tree for variable `a` is simply `TEMP a`).

   (a) `a<b`

   Answer:

```
ESEQ(SEQ(MOVE(TEMP t, CONST 1),
         SEQ(CJUMP(LT, TEMP a, TEMP b, L0, L1),
             SEQ(LABEL L1,
                 SEQ(MOVE(TEMP t, CONST 0),
                     LABEL L0)))),
     TEMP t)
```

(b) `if a<b then c:=a else c:=b`, translated using the tree from part 4a.

Answer:

```
SEQ(SEQ(CJUMP(NE,
              ESEQ(SEQ(MOVE(TEMP t, CONST 1),
                       SEQ(CJUMP(LT, TEMP a, TEMP b, L0, L1),
                           SEQ(LABEL L1,
                               SEQ(MOVE(TEMP t, CONST 0),
                                   LABEL L0)))),
                   TEMP t),
              CONST 0,
              L2,
              L3),
        SEQ(SEQ(SEQ(LABEL L2, MOVE(TEMP c, TEMP a)),
                JUMP(NAME L4)),
            SEQ(SEQ(LABEL L3, MOVE(TEMP c, TEMP b)),
                JUMP(NAME L4)))),
    LABEL L4)
```

(c) `if a<b then c:=a else c:=b`, translated in a less clumsy way

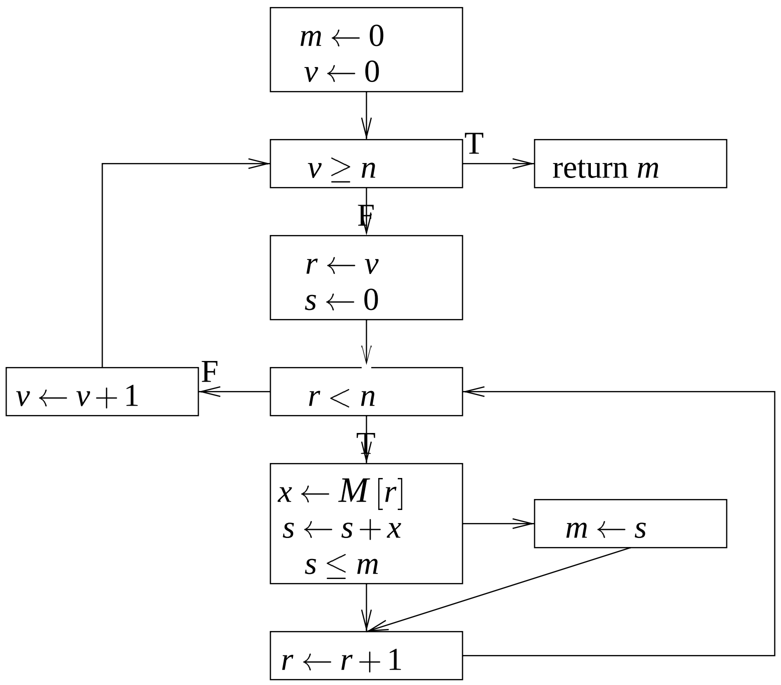
Answer:

```
SEQ(SEQ(CJUMP(LT, TEMP a, TEMP b, L0, L1),
        SEQ(SEQ(SEQ(LABEL L0,
                    MOVE(TEMP c, TEMP a)),
                JUMP(NAME L2)),
            SEQ(SEQ(LABEL L1,
                    MOVE(TEMP c, TEMP b)),
                JUMP(NAME L2)))),
    LABEL L2)
```

5. (Basic blocks and traces; 10%) Consider the following program:

```
1   $m \leftarrow 0$
2   $v \leftarrow 0$
3   **if** $v \geq n$ **goto** 15
4   $r \leftarrow v$
5   $s \leftarrow 0$
6   **if** $r < n$ **goto** 9
7   $v \leftarrow v + 1$
8   **goto** 3
9   $x \leftarrow M[r]$
10  $s \leftarrow s + x$
11  **if** $s \leq m$ **goto** 13
12  $m \leftarrow s$
13  $r \leftarrow r + 1$
14  **goto** 6
15  **return** $m$
```

Break this program into basic blocks, and draw its control flow graph. **Answer:**



6. (Instruction selection; 10%) For each of the following expressions, draw the IR tree and generate MIPS instructions using "maximal munch" (you need not eliminate the frame pointer `fp`). Circle the tiles and number them *in the order that they are "munched"* (i.e., the order that instructions are emitted for the tile).

   (a)
   ```
   MOVE(MEM(BINOP(PLUS,
                  BINOP(PLUS, CONST 1000, MEM(TEMP x)),
                  TEMP fp)),
        CONST 0)
   ```

   Answer:

   ```
   [3 = MOVE(MEM(BINOP(PLUS,
                       [2 = BINOP(PLUS, CONST 1000, [1 = MEM(TEMP x)])],
                       TEMP fp)),
             CONST 0)]
   ```

   ```
   lw  t1 0(x)
   add t2 t1 1000
   sw  zero t2(fp)
   ```

   (b) `BINOP(MUL, CONST 5, MEM(CONST 100))`

   Answer:

   ```
   [2 = BINOP(MUL, CONST 5, [1 = MEM(CONST 100)])]
   ```

   ```
   lw  t1 100(zero)
   mul t2 t1 5
   ```