CS 352 — Compilers: Principles and Practice
Final Examination, 12/16/11

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. (LL parsing, context-free grammars; 30%) Consider the following simple context free grammars:

Grammar G_1	Grammar G_2	Grammar G_3
$S \rightarrow A$	$S \rightarrow A$	$S \rightarrow A$
$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
$A \rightarrow bbA$	$A \rightarrow bAb$	$A \rightarrow Abb$

Let $L = L(G_1) = L(G_2) = L(G_3)$ (these grammars all generate the same language).

(a) (10%) Construct the LL(1) parse table for G_1. Is G_1 in the class LL(1)? Explain.

Answer:

Yes, here is its LL(1) parse table:

$\textsc{first}(S) = \{b, \varepsilon\}$
$\textsc{first}(A) = \{b, \varepsilon\}$

$\textsc{follow}(S) = \{\$\}$
$\textsc{follow}(A) = \{\$\}$

LL(1) parse table

	b	$\$$
S	$S \rightarrow A$	$S \rightarrow A$
A	$A \rightarrow bbA$	$A \rightarrow \varepsilon$

Since there are no multiply defined entries in its LL(1) parse table G_1 is LL(1).

(b) (5%) Is G_2 LL(1)? Explain.

Answer:

G_2 is not LL(1) since b predicts both $A \rightarrow \varepsilon$ and $A \rightarrow bAb$.

(c) (5%) Is G_3 LL(1)? Explain.

Answer:

G_3 is not LL(1) since it is left-recursive.

(d) (5%) Describe the language L.

Answer:

Strings with an even number of bs (including 0 of them).

(e) (5%) Of the language classes we have discussed in this course, what is the smallest category into which L fits? Explain.

Answer:

The language is regular. It is defined by the following regular expression: $(bb)^*$.

2. (Runtime management; 20%) Consider the following `cj` program as it executes. Assume that `puts` prints its string argument to the console followed by a new line, and that `putint` prints its integer argument.

```
def puts(t: text); /* print t followed by the new line character */
def putint(i: int); /* print i (without a new line character) */
{
 type node = record { value: int; left, right: nodePtr };
 type nodePtr = ref node;
 def insert(n: nodePtr; root: nodePtr): nodePtr {
  if root == nil then return n;
  if n.value <= root.value then
   root.left := insert(n, root.left);
  else
    root.right := insert(n, root.right);
  return root;
 }
 def pt(n: ref node) {
  if n == nil then return;
  pt(n.left);
  putint(n.value); /* print the integer n.value */
  puts(""); /* print the new line character */
  pt(n.right);
 }
 var tree: nodePtr := nil;
 var n: nodePtr;
 n := new ref node; n.value := 45; tree := insert(n, tree);
 n := new ref node; n.value := 33; tree := insert(n, tree);
 n := new ref node; n.value := 24; tree := insert(n, tree);
 n := new ref node; n.value := 22; tree := insert(n, tree);
 n := new ref node; n.value := 22; tree := insert(n, tree);
 n := new ref node; n.value := 10; tree := insert(n, tree);
 pt(tree);
}
```

(a) (5%) What output does this program produce?

Answer:

```
10
22
22
24
33
45
```

(b) (15%) Show a diagram of MIPS stack frames at the point when this program is executing the call to `pt` and has just printed out the string `10` to the console at line 17. Show where *all* the local variables *and* heap variables are (assume that all local variables are stored in memory in the stack, not in registers); show the value of all integer variables in the stack *and heap*, as well as each variable in the stack *and heap* containing a reference to the heap, and the object it refers to.

Answer:

```
         STACK                                     HEAP


       +---------+
main:  | tree  --+----------+---------------------> 45
       | ...     |          |                      /  \
       +---------+          | +-----------------> 33  null
pt:    | n     --+----------+ |                  /  \
       | ...     |            | +-------------> 24  null
       +---------+            | |              /  \
pt:    | n     --+------------+ | +---------> 22  null
       | ...     |              | |          /  \
       +---------+              | | +-----> 22  null
pt:    | n     --+--------------+ | |      /  \
       | ...     |                | | +-> 10  null
       +---------+                | | |
pt:    | n     --+----------------+ | |
       | ...     |                  | |
       +---------+                  | |
pt:    | n     --+------------------+ |
       | ...     |                    |
       +---------+                    |
pt:    | n     --+--------------------+
       | ...     |
       +---------+
```

3. (Translation to intermediate code; 10%) Consider the following `cj` program.

```
var a: array [5] of int;
{
 for i := 0 to 4 do
  a[i] := i;
}
```

Give IR trees for this program. You can assume that the global variable `a` is allocated statically at location `NAME a`, and that it requires no initialization. Similarly, you can refer to variable `i` as a temporary using `TEMP i`. Also, because the bounds of the **`for`** loop index `i` are within the bounds of the array there is no need to check that `i` is in range.

Answer:

```
MOVE(
 TEMP t.0,
 CONST 0),
MOVE(
 TEMP t.1,
 CONST 4),
MOVE(
 TEMP t.2,
 CONST 1),
LABEL L.0,
MOVE(
 TEMP i,
 TEMP t.0),
MOVE(
 MEM(
  ADD(
   NAME a,
   MUL(
    TEMP i,
    CONST 4)),
  CONST 0, 4),
 TEMP i),
MOVE(
 TEMP t.0,
 ADD(
  TEMP t.0,
  TEMP t.2)),
LABEL L.1,
BLE(
 TEMP t.0,
 TEMP t.1,
 L.0, L.2),
LABEL L.2
```

4. (Static links; 10%) Consider the following `cj` program:

```
def p() {
  def q() {
    def r() {
      ...
    }
    ...
  }
  ...
}
```

Assume the following call sequence: `p`, `q`, `r`, `q`, `p`, `q`, `r`, `q`, `r`. Draw the call stack at the time of the last invocation of `r`, and show the static link for each invocation as a pointer from the stack frame to its outer scope frame.

Answer:

```
p <-+ <-----+
    |       |
q --+ <-+   |
        |   |
r ------+   |
            |
q ----------+

p <-+ <-----+
    |       |
q --+ <-+   |
        |   |
r ------+   |
            |
q ----------+ <-+
                |
r --------------+
```

5. (Scoping and visibility; 15%) Shown below is a `cj` program. At runtime, static links are used to allow functions to access non-local variables. In the program, some variable names have been replaced with boxes.

```
{
    var □: int;
    var □: int;
    def F1() {
        var □: int;
        if x == 0 then z := 10;
    }
    def F2(): int {
        var □: int;
        def F3() {
            var □: int;
            x := 0;
            y := 2 * y;
            z := y - x;
        }
        □ := 20;
        F3();
        return x + y;
    }
    □ := 10;
    □ := 0;
    F1();
    F2();
}
```

Fill in the boxes so that the program includes no use of an undeclared or uninitialized variable, and so that the program is consistent with the following clues [Hint: use the clues in the order in which they are given]:

(a) The activation record with space for the variable `y` used in `F3` is found by following two access links.

(b) The activation record with space for the variable `x` used in `F3` is found by following one access link.

(c) `F1` does not use any uninitialized or undeclared variables.

(d) The value returned by `F2` is `20`.

6. (Parameter passing; 15%) Consider the following `cj` program, where *mode* is some parameter passing mode.

```
{
    var z;
    def foo(mode x, y: int) {
        x := x + y;
        z := z + x + y;
    }
    z := 5;
    foo(z, z);
}
```

For each of the following scenarios, what is the value of `z` at the end of the program's execution?

(a) (5%) All parameters are passed by *value* (*ie*, *mode* is **`val`**).

Answer:

20

(b) (5%) All parameters are passed by *reference* (*ie*, *mode* is **`var`**).

Answer:

30

(c) (5%) All parameters are passed as *value-result*, also called *copy-in/copy-out* (*ie*, *mode* is **`inout`**).

Answer:

5