CS 345 - Programming Languages
Fall 2010

MIDTERM #1

October 5, 2010

DO NOT OPEN UNTIL INSTRUCTED

YOUR NAME: _____

Collaboration policy

No collaboration is permitted on this midterm. Any cheating (*e.g.*, submitting another person's work as your own, or permitting your work to be copied) will automatically result in a failing grade. The Computer Sciences department code of conduct can be found at `http://www.cs.utexas.edu/academics/conduct/`.

Midterm #1 (85 points)

Problem 1 (15 points)

Circle only one of the choices **(3 points each)**.

1. **TRUE** **FALSE** Algol 60 was the first language to introduce lexical block scoping.
2. **TRUE** **FALSE** A shift-reduce conflict can only occur if the grammar is ambiguous.
3. **TRUE** **FALSE** The lifetime of a variable is equal to the lifetime of the activation record corresponding to its scope.
4. **TRUE** **FALSE** Closures are necessary in dynamically scoped languages to keep track of each function's scope.
5. **TRUE** **FALSE** Finding an exception handler in ML simply requires walking up the stack from where the exception occurred.

Problem 2 (15 points)

Define the following terms:

Ambiguous grammar:

R-value of a variable:

Scope of a variable:

Pass-by-reference:

Access link:

Problem 3

Problem 3a (3 points)

What is the purpose of *lexical analysis*? What type of abstract machine is used to implement it?

Problem 3b (3 points)

What is the purpose of *syntax analysis*? What type of abstract machine is used to implement it?

Problem 4

Problem 4a (3 points)

What is the "dangling else" problem?

Problem 4b (8 points)

Write a grammar of `if-then-else` expressions where every `if` statement must be terminated by `endif`. Does this grammar solve the "dangling else" problem? Explain your answer in terms of shift-reduce conflicts.

Problem 4c (3 points)

Can your new grammar be recognized by a deterministic finite automaton? Explain.

Problem 5

Explain the evaluation of the following statements in terms of l-values and r-values.

Problem 5a (3 points)

```
int x = 1;
int a[5] = {5,4,3,2,1};
```

Problem 5b (3 points)

```
int *p = a;
```

Problem 5c (3 points)

```
int **q = &p;
```

Problem 5d (3 points)

```
a[3] = a[x+1] * **q;
```

Problem 6

Consider the following program written in Algol-like pseudocode:

```
begin
  integer i;

  procedure foo(integer x, integer y);
     begin
        x := x+1;
        y := x+1;
        x := y;
        i := i+1;
     end

  i := 1;
  foo(i,i);
  print i;
end
```

What would this program print under each of the following parameter passing mechanisms:

Problem 6a (3 points)

Pass-by-value:

Problem 6a (3 points)

Pass-by-reference:

Problem 6a (3 points)

Pass-by-value-result:

Problem 7

In ML and most other functional languages, it is legal to declare a "local function," *i.e.*, a function defined within the scope of another function. For example, in ML you might write:

```
fun f() = let val i = 1
              fun g() = i
          in
              (print(Int.toString( g() ));
              g)
```

```
        end;
print(Int.toString( f()() ));
```

This program declares a function `f` which declares a local variable `i` and a local function `g`. The function `g` simply returns the value of `i`. When you execute this program, it prints 1 twice.

Because `f` returns a function, its scope must remain "alive" even after `f` finished its execution. As we discussed in class, ML solves this problem by placing both the activation record for the call to `f` and the closure for `g` on the heap.

In ANSI C, there are no local functions, so there is no way to write an equivalent program. GNU CC (GCC) compiler, however, does allow local function declarations. Here is how to write an equivalent program in GNU C:

```
#include <stdio.h>
typedef int (*fn_t)();
fn_t f() {
   int i=1;
   int g() { return i; }
   printf("%d\n", g());
   return &g;
}
int main() {
   printf("%d\n", (*f())() );
}
```

GCC compiles local functions in the usual way, except that references to the activation record of an enclosing function are done via a static (access) link, like in ML.

A particular instance of a local function is a piece of code (called the *trampoline*) placed on the stack, that sets the static chain and jumps to the beginning of code for the compiled function. The trampoline serves the same purpose as a closure.

Unlike ML, however, GCC places both activation records and trampolines on the stack and makes no specific effort to solve the problem of keeping the scope "alive" after the function returns.

Problem 7a (5 points)

The output of the GNU C program above is

```
1
-1073743424
```

Explain why this program does not print 1 twice, as one might expect. Where does the second number come from?

Problem 7b (3 points)

Why does ML deviate from stack ("last-in-first-out") storage management for closures and activation records?

Problem 8

ML has functions `hd` and `tl` to return, respectively, the head (*i.e.*, the first element) and the tail (*i.e.*, the remaining elements) of a list. Both functions raise an exception `Empty` if the list is empty.

Suppose that we redefine these functions so that they raise, respectively, exceptions `Hd` and `Tl` when applied to an empty list. The behavior of non-empty lists remains the same.

```
- hd(nil);
uncaught exception Hd
- tl(nil);
uncaught exception Tl
```

Now consider the following "identity" function on lists:

```
fun id(l) = hd(l)::tl(l) handle Hd => nil;
```

Problem 8a (3 points)

What is the result of evaluating `id(nil)`? Why?

Problem 8b (3 points)

Does the function `id` need to handle exception `Tl`? Why?