

CS 345 - Programming Languages
Spring 2008

MIDTERM #1

February 26, 2008

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/users/ear/CodeOfConduct.html>

Midterm #1 (75 points)

Problem 1 (12 points)

Define the following terms:

Local variable:

Access link:

Run-time type error:

Polymorphic function:

Problem 2

Problem 2a (4 points)

What is the “dangling else” problem?

Problem 2b (7 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 2c (4 points)

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

Problem 3

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

Problem 3a (3 points)

```
int x = 0;
```

Problem 3b (3 points)

```
int *p = &x;
```

Problem 3c (3 points)

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

Problem 3d (3 points)

```
*p++;
```

Problem 4

Consider the following program:

```
void swap(int[] list, int i, int j) {  
    int temp = list[i];  
    list[i] = list[j];  
    list[j] = temp;  
}  
  
void main() {  
    int x[3] = {5, 2, 4};  
    swap(x,i,j);  
}
```

What is the final value of the array `x` for each of the following parameter passing assumptions:

Problem 4a (2 points)

Argument `x` is passed by value.

Problem 4b (2 points)

Argument `x` is passed by reference.

Problem 4c (2 points)

Argument `x` is passed by value-result.

Problem 5 (5 points)

Why is tail recursion elimination useful?

Problem 6

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 = 4
           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 4 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=4;
    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 6a (5 points)

The output of the GNU C program above is

```

4
-1073743424

```

Explain why this program does not print 4 twice, as one might expect. Where does the second number come from?

Problem 6b (5 points)

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

Problem 6c (4 points)

What might be some advantages of placing trampolines and activation records on the stack, even when local functions are used?

Problem 6d (4 points)

Do you think the decision that the GNU C designers made, namely, to place trampolines and activation records on the stack, is consistent with the basic design goals of C? Why or why not?

Problem 7 (7 points)

Draw the parse graph and use it to perform type inference for the following ML function:

```
fun arith(f,g,x) = f(5 + g(x))
```