Bryan Kline

CS326

Homework 3

LaTex (TeXstudio)

10/11/2016

1. Using the C++ programming language, indicate the binding time (language design, language implementation, compilation, link, run, etc.) for each of the following attributes. Justify your answer.

a) The variable declaration that corresponds to a certain variable reference (use)
This takes place during compile time, as C and C++ are statically scoped.

b) The range of possible values for integer numbers
This is determined at during the language implementation.

c) The meaning of char
This is determined when the language is initially designed.

d) The address of a local variable
This depends on which address is meant, if it's the physical address then it's during load time, and if it's the virtual address then it's during compile time.

e) The address of a library function
This occurs at link time.

f) The referencing environment of a function passed as a parameter
This is done at compile time, as C and C++ don't have nested functions.

g) The total amount of memory needed by the data in a program
This is determined at runtime.

2. Can a language that uses dynamic scoping do type checking at compile time? Why? Can a language that uses static scoping do type checking at run time? Why?

No, a language that uses dynamic scoping cannot do type checking at compile time. This is because if the scope is determined dynamically, and if the flow of control of a program isn't something that can be predicted ahead of time, i.e. what functions might be called in what order, then types cannot be checked because it won't know what to expect. Also, a language that uses static scoping cannot do type checking at runtime because that is all done at compile time.

3. Does Scheme use static or dynamic scoping? Write a short Scheme program that proves your answer.

Scheme uses static scoping, as the textbook for this class points out (p. 140) as a footnote to listings dynamically scoped languages. The following Scheme code snippet will show this:

```
(define x 0)
(define (function)
        (set! x 1))

(set! x 2)
(function)
(display x)
```

The output is 1, not 2, which demonstrates that the scoping is static because the global x is set to 1 by the call to function, meaning that the next scope out from function, the global scope, was changed.

4. Consider the following pseudo-code:

```
x : integer;       - - global
procedure set_x (n : integer)
x := n;
procedure print_x
    write_integer (x);
procedure foo (S, P : procedure; n : integer)
```

```
    x : integer;
    if n in 1,3
        set_x(n);
    else
        S(n);
    if n in 1,2
        print_x;
    else
        P;
- - main program
set_x(0); foo (set_x, print_x, 1); print_x;
set_x(0); foo (set_x, print_x, 2); print_x;
set_x(0); foo (set_x, print_x, 3); print_x;
set_x(0); foo (set_x, print_x, 4); print_x;
```

Assume that the language uses dynamic scoping. What does this program print if the language uses shallow binding? Why? What does it print with deep binding? Why? Note: At exactly one point during execution in the deep binding case, the program will attempt to print an uninitialized variable. Simply write a ? for the value printed at that point.

NOTE: Newline characters, while never printed in the code, are added for each line for clarity.

Shallow binding:

```
1 0
2 0
3 0
4 0
```

The above is printed out if the binding is shallow because the reference environment comes from whenever the function is called, in this case when set_x and print_x are called in foo, not from when they were passed into foo, so they print the x from inside foo.

Deep binding:

```
1 0
? 2
0 0
4 4
```

The above is printed out if the binding is deep because the reference environment comes from when the functions are passed into foo, and so if in foo the local set_x and print_x functions are called then the local x is printed, otherwise it's the global one, and when it leaves foo it's also the global x that's printed.