Midterm: CS4215 Programming Language Implementation

2/3/2012

You have 45 minutes to complete the exam. Use a B2 pencil to fill up the provided MCQ form. Leave Section A blank. Fill up Sections B and C.

After finishing, place the MCQ sheet on top of the question sheet and leave both on the table, when you exit the room.

Question 1: Which one of the following statements about compilers is *false*?

1 [A] A compiler always produces a program, never the result of executing a program.

1 [B] A compiler may execute instructions of the given program such as arithmetic operations, as long as the meaning of the output program is not changed.

1 [C] A compiler needs to run the given program in the from-language in order to produce a program in the to-language.

1 [D] A compiler may avoid translating parts of a program, if these parts will never be executed.

1 [E] A correctly implemented compiler never runs into an infinite loop.

Answer 1:

1 [C] A compiler does not need to run the given program. It needs to translate the instructions given in the program to the to-language.

The other statements are true. Regarding D, a compiler may simply ignore the `else` part of a conditional, if it is clear at compile time that the condition of the conditional is always true. Regarding E, a compiler needs to terminate, regardless of its input.

Question 2: Which one of the following statements about ePL is false?

2 [A] The execution of ePL programs according to the dynamic semantics always terminates.

2 [B] Some ePL programs are not well typed, but their execution according to the dynamic semantics produces a value.

2 [C] Some ePL programs are well-typed, but their execution according to the dynamic semantics does not produce a value.

2 [D] ePL programs that are well-typed and contain the symbol `=` will produce a boolean value.

2 [E] ePL programs that are well-typed and do not contain integer constants will produce a boolean value.

Answer 2:

2 [B] This never happens, because the transition relation of ePL always gets stuck when the given program is well-typed.

2 [D] A counter-example for this is

```
1/0 = 2
```

This program contains `=` but does not produce a boolean value, according to the semantics covered.

All other statements are true. Regarding C: This is of course division by zero.

Question 3: What is the result of evaluating the following simPL program according to the dynamic semantics given in class?

```
(fun {(int -> bool) * int -> bool} x y -> (x y) end
 fun {int -> bool} x -> x > 0 end
 4)
```

3 [A] `fun {int -> bool} x -> x > 0 end`

3 [B] `0`

3 [C] `4`

3 [D] `true`

3 [E] `false`

Answer 3:

3 [D] of course

Question 4: Which statement about the following simPL program is correct?

```
let {int} x = 1
    {int -> int} y = fun {int -> int} y -> x * y end
in  {int} (y x)
end
```

4 [A] The program is well-typed and its evaluation according to dynamic semantics produces an integer value.

4 [B] The program is well-typed but its evaluation according to dynamic semantics produces a function value

4 [C] The program is well-typed and its evaluation according to dynamic semantics produces a boolean value instead of an integer.

4 [D] The program is not well-typed.

4 [E] None of the above.

Answer 4:

4 [D] The `let` statement is an abbreviation for

```
(fun {int * (int -> int) -> int} x y -> (y x) end
 1
 fun {int -> int} y -> x * y end
)
```

Here you can clearly see that the last `y` in this program is a free variable. Thus the program is not well-typed.

Question 5: Consider the "high-level" virtual machine covered in Lab Task Week 5, in which Java's `Stack` class is used for operand stacks and the runtime stack. Of course, in practice, such stacks are always finite, and their maximal size is determined by the memory available to the JVM. Which statement about this machine is false?

5 [A] When running the result of compiling a well-typed program, the machine may encounter a runtime stack overflow.

5 [B] When running an sVML program, the machine may encounter an operand stack overflow.

5 [C] Without running the program, we can predict the maximal size of any operand stack for any well-typed program.

5 [D] Without running the program, we can predict the maximal size of the runtime stack for any well-typed program.

5 [E] Without running the program, we can predict the size that the runtime stack will have after running any well-typed program program.

Answer 5:

5 [D] It is undecidable whether an arbitrary given program terminates or not. It is also undecidable whether a given arbitrary application of a boolean function returns `true` or not. So here is a program for which we cannot predict whether it will produce a runtime stack overflow:

```
let {int -> bool} randomboolean = fun {int -> bool} x -> ... end
in {int}
   if (randomboolean 0) then 1
   else (recfun loop {int -> int} x -> (loop x) end 2)
   end
end
```

All other statements are true. Regarding C: This was discussed in the labs. Remember that every Java compiler does this, and our latest simPL compiler also does it. Regarding E: The size of the runtime stack will be 0, because in order to terminate, the program needs to return from every function call.

Question 6: A stack encounters an "underflow", when the stack is empty and an attempt to pop an element is made. Which statement is false?

6 [A] The result of compiling a well-typed program will never encounter an operand stack underflow.

6 [B] The result of compiling a well-typed program will never encounter an runtime stack underflow.

6 [C] It is impossible to write an sVML program that encounters a runtime stack underflow.

6 [D] We can write an sVML program with one instruction that encounters an operand stack underflow.

6 [E] By analyzing a given sVML program, we cannot always predict whether a runtime stack underflow will happen at runtime or not.

Answer 6:

6 [C] Example: `RETURN`

All other alternatives are true. Regarding D: Example: `DONE`. Regarding E: undecidability.

Question 7: Which of the following statements about typing simPL programs is true?

7 [A] Well-typed simPL programs whose type is `bool` may compute an integer value according to the dynamics semantics.

7 [B] Well-typed simPL programs whose type is `int` may compute a boolean value according to the dynamics semantics.

7 [C] Well-typed simPL programs whose type is `bool` may compute a function value according to the dynamics semantics.

7 [D] Well-typed simPL programs whose type is `int` may compute a function value according to the dynamics semantics.

7 [E] None of the above.

Answer 7:

7 [E] of course. That's the point of typing.

Question 8: Consider the static semantics of simPL as introduced in the lectures. How many type expressions T exist such that

```
fun {T} x -> x end
```

is well-typed?

8 [A] 0

8 [B] 1

8 [C] 2

8 [D] 3

8 [E] infinitely many

Answer 8:

8 [E] Any type will do the job.

Question 9: Consider the static semantics of simPL as introduced in the lectures. How many type expressions T exist such that

```
fun {T} x -> (x x) end
```

is well-typed?

9 [A] 0

9 [B] 1

9 [C] 2

9 [D] 3

9 [E] infinitely many

Answer 9:

9 [A] Explanation in the lab

Question 10: Consider the static semantics of simPL as introduced in the lectures. How many type expressions T exist such that

```
fun {T} x -> (x 1) end
```

is well-typed?

10 [A] 0

10 [B] 1

10 [C] 2

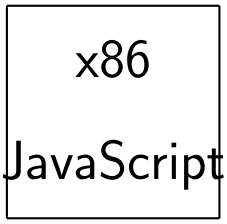
10 [D] 3

10 [E] infinitely many

Answer 10:

10 [E] Any function type with one argument type whose argument type is `int` can be used.

Question 11: Consider the following T-diagram of an interpreter.



Which one of the following statements is true?

11 [A] This interpreter is impossible to execute on an x86 machine, because of circularity.

11 [B] This interpreter produces x86 machine code when run on an interpreter for JavaScript.

11 [C] This interpreter produces JavaScript code when run on an interpreter for x86 machine code (such as x86 hardware).

11 [D] This interpreter can run x86 machine code without x86 hardware, when an interpreter for JavaScript is available.

11 [E] This interpreter is easy to implement if we have x86 hardware. We only need to execute each instruction in terms of itself.

Answer 11:

11 [D] of course.

All other alternatives are false.

Question 12: The denotational semantics for simPL distinguishes between expressible values and denotable values. Which of the following statements is true?

12 [A] Some values are expressible, but not denotable.

12 [B] Some values are denotable, but not expressible.

12 [C] Function values are neither expressible nor denotable.

12 [D] Some values are denotable, but they can never be the value of an indentifier in an environment.

12 [E] Some values are expressible, but they can never be the result of evaluating a simPL program, according to its denotational semantics.

Answer 12:

12 [A] The error value is expressible (it can be the result of evaluating an expression), but not denotable (no identifier will ever refer to the error value during evaluation).

All other alternatives are false.

Question 13: Consider the possibility of implementing a type checker for sVML. Which one of the following statements it false. (A stack "underflow" happens when the stack is empty and a pop operation is attempted.)

13 [A] Such a type checker could make sure that `PLUS` always finds two integers on the operand stack.

13 [B] Such a type checker could compute a safe limit for any operand stack, and thus prevent operand stack overflow.

13 [C] Such a type checker should always admit the result of compiling well-typed simPL programs into sMVL.

13 [D] Such a type checker should reject all programs that will encounter a runtime stack overflow.

13 [E] Such a type checker should prevent any runtime stack underflow.

Answer 13:

13 [D] The type checker cannot do this, due to undecidability.

All other statements are true.

Question 14: Consider running a program using Cheney's algorithm for garbage collection using a machine whose heap size is 100 MB. The `flip` function is called three times, and each time, the residency r is 20%. Immediately after the last run of `flip` we know that the mutator program has so far allocated about

14 [A] 120 MB.

14 [B] 150 MB.

14 [C] 210 MB.

14 [D] 240 MB.

14 [E] 300 MB.

Answer 14:

14 [A] The correct number is 110 MB, which is "about" 120 MB. So students who answered A will get the point. Before (and of course also after) the first run of `flip` the mutator has allocated 50 MB. Before the second run the mutator has allocated another 30 MB, and before the third run the mutator has allocated another 30 MB. Altogether 110 MB.

Question 15: Which one of the following statements is correct?

15 [A] Cheney's algorithm traverses the live memory in a breadth-first manner.

15 [B] Cheney's algorithm visits every dead node exactly once.

15 [C] Cheney's algorithm cannot reclaim cyclic data structures.

15 [D] Cheney's algorithm is an example of an incremental memory management technique.

15 [E] Cheney's algorithm maintains a free list where un-used nodes are kept.

Answer 15:

15 [A] The to-space forms a queue, implemented by the pointers `scan` and `free`. This leads to a breadth-first traversal of live memory.

All other statements are false.