

THE UNIVERSITY OF NEW SOUTH WALES

Mid-Session Sample Exam Solutions

Session 1 2007

COMP3161/COMP9161

Concepts of Programming Languages

Markers: answers are provided in bold font. *Additional; information which is not necessarily expected to be part of the student's answer is given in this font.*

Question 1 [2 Marks]

Consider the following expression e :

```
let x = 5 in
  let x = 10 + x in
    let y = 5 + x in
      x + let x = 15 in x * y end
    end
  end
end
```

Find an expression e' which is α -equivalent to e where every variable is only used at a single binding position (i.e., every variable name in the expression occurs only once in a left-hand side of a let-binding).

Many different solutions possible.

```
let x = 5 in let x1 = 10 + x in let y = 5 + x1 in x1 + let x2 = 15 in
x2 * y end end end end
```

Question 2 [6 Marks]

In the lecture we discussed the role of the lexer, the parser, and the (static) semantic analyser. For each of these components, give an example of a program error that can be detected by that component. (The error may be in terms of C, Haskell or the language of arithmetic expressions with let-bindings discussed in the lecture).

- **lexer:** decomposes the program string into a sequence of tokens. Possible errors it can detect: if a substring is neither a keyword, identifier, nor a constant value of any type. For example, in C, `4rewrcs` would cause an error.
- **parser:** analyses the structure of a program. Possible errors it can detect: if expressions or definitions are not well-formed, e.g. in Haskell, an if-expression without a then-branch, mismatching parens in any language.
- **static semantic analysis:** checks static semantic properties, for example, if all variables are in scope, no type errors occur.

Question 3 [6 Marks]

What is the difference between concrete and abstract syntax of a programming language? (keep your answer brief — it may be easiest to describe the difference if you use the possible concrete and abstract syntax of an actual language construct as an example)

The concrete syntax of a language is designed with the human user in mind (at least it should be :-), so it has operations in infix notation, symbols which only serve to denote the end of blocks, expressions, or statements, like parenthesis', semicolons, and the like.

Abstract syntax are terms designed to store the information which is necessary for the rest of the translation process. For example, for arithmetic expressions, we have brackets in the concrete syntax to avoid any ambiguity, and allow infix notation, as it is more convenient to use, eg, $2 * (3+4)$. The abstract syntax for the expression is simply a nested term `times (2, plus(3,4))`

Question 4 [6 Marks]

Given the following inference rules which define the big step semantics for algebraic expression in abstract syntax. What is the value of

`plus (num(5), let (num(7), x. plus (x, num(1)))`

Give the derivation and annotate each rule application with the number of the rule you used.

$$(1) \frac{}{\text{num}(n) \Downarrow \text{num}(n)}$$

$$(2) \frac{e_1 \Downarrow \text{num}(n_1) \quad e_2 \Downarrow \text{num}(n_2)}{\text{plus}(e_1, e_2) \Downarrow \text{num}(n_1 + n_2)}$$

$$(3) \frac{e_1 \Downarrow \text{num}(n_1) \quad e_2 \Downarrow \text{num}(n_2)}{\text{times}(e_1, e_2) \Downarrow \text{num}(n_1 * n_2)}$$

$$(4) \frac{e_1 \Downarrow \text{num}(n_1) \quad \{\text{num}(n_1)/x\}e_2 \Downarrow \text{num}(n_2)}{\text{let}(e_1, x.e_2) \Downarrow \text{num}(n_2)}$$

Question 5 [20 Marks]

Given a set of induction rules defining *nat*

$$(1) \frac{}{0 \text{ nat}}$$

$$(2) \frac{n \text{ nat}}{s(n) \text{ nat}}$$

and a set of rules defining *enat*:

$$(3) \frac{}{0 \text{ enat}}$$

$$(4) \frac{}{s(0) \text{ enat}}$$

$$(5) \frac{n \text{ enat}}{s(s(n)) \text{ enat}}$$

Using rule induction, show that both definitions are equivalent, that is, for all x , $x \text{ nat}$ is derivable if and only if $x \text{ enat}$ is derivable. Clearly state which cases you have to consider, and what the induction hypothesis is for each case. Annotate derivations with the number of the rule you used.

Hint: One direction of the proof is straight forward. For the other direction, you might find it helpful to first prove a Lemma.

This is just one possibility, following the pattern of proofs we discussed in the lecture.

We have to show that

A) if $x \text{ enat}$ then $x \text{ nat}$: we have to consider three cases:

i) $x = 0$

$$(1) \frac{}{0 \text{ nat}}$$

ii) $x = s(0)$

$$(2) \frac{1 \frac{}{0 \text{ nat}}}{s(0) \text{ nat}}$$

iii) $x = s(s(x'))$ for $x' \text{ enat}$

I.H.: if $x' \text{ enat}$ then $x' \text{ nat}$

$$(2) \frac{(I.H.) \frac{}{x' \text{ nat}}}{s(x') \text{ nat}} \\ (2) \frac{}{s(s(x')) \text{ nat}}$$

B) if $x \text{ nat}$ then $x \text{ enat}$

Lemma:

$$\frac{n \text{ enat}}{s(n) \text{ enat}}$$

Proof of Lemma: three cases to consider:

- $n = 0$: **rule (3)** $s(0) \text{ enat}$
- $n = s(0)$: **rule (4)** $s(s(0)) \text{ enat}$
- $n = s(s(n'))$ for **(a1)**: $n' \text{ enat}$
I.H. if $n' \text{ enat}$ then $s(n') \text{ enat}$

$$(5) \frac{(I.H.) \frac{(a1) \frac{}{n' \text{ nat}}}{s(n) \text{ enat}}}{s(s(s(n))) \text{ enat}}$$

Now back to the proof goal: if $x \text{ nat}$ then $x \text{ enat}$

i) $x = 0$

ii) $x = s(x')$ with $x' \text{ nat}$

I.H. if $x' \text{ nat}$ then $x' \text{ enat}$

Proof: I.H., Lemma