THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

2007 A SEMESTER EXAMINATIONS

DEPARTMENT	Department of Computer Science
PAPER TITLE	Programming Languages
TIME ALLOWED	Three Hours
NUMBER OF QUESTIONS IN PAPER	Six
NUMBER OF QUESTIONS TO BE ANSWERED	Six
VALUE OF EACH QUESTION	The value of each question is indicated. The total number of marks achievable is 100.
GENERAL INSTRUCTIONS	Answer ALL SIX questions.
SPECIAL INSTRUCTIONS	None
CALCULATORS PERMITTED	No

The Appendix at the end of this paper contains some definitions that you might find useful when answering questions 1 to 3.

Question 1

a) Say what types, in Haskell, the following expressions have:

(i) $(+)$

(ii) $(1+)$

(iii) $(1+2)$

(iv) $map\ (1+)\ [1,2,3]$

(v) $map\ (1+)$

(vi) $map\ (\backslash\ x\!-\!>(\prime a\prime, x))[1,2,3]$

(6 marks)

b) Say what values, in Haskell, the following expressions have:

(i) $map\ (1+)\ [1,2,3]$

(ii) $(map\ (\backslash\ x- > (\prime a\prime, x))\ [1,2,3]$

(iii) $f\ (Rectangle\ 6.0\ 7.0)\ 1.0\ 2.0$

$$
\begin{aligned}
where\ f\ s\ dx\ dy = {} & case\ s\ of \\
& RtTriangle\ s1\ s2 = RtTriangle\ s1 * dx\ s2 * dy \\
& Rectangle\ s1\ s2 = Rectangle\ s1 * dx\ s2 * dy \\
& Ellipse\ r1\ r2 = Ellipse\ r1 * dx\ r2 * dy
\end{aligned}
$$

given the data declaration

$$
\begin{aligned}
data\ Shape = {} & Rectangle\ Float\ Float\ | \\
& Ellipse\ Float\ Float\ | \\
& RtTriangle\ Float\ Float
\end{aligned}
$$

(4 marks)

Answer:

a) *(i)* $Integer\ -> Integer\ -> Integer$
A better answer is $Num\ a => a- > a- > a$ *but most won't give that as we did not cover type classes.*

(ii) $Integer\ -> Integer$

(iii) $Integer$

(iv) $[Integer]$

(v) $[Integer]\ -> [Integer]$

(vi) $[(Char, Integer)]$

b) *(i)* $[2,3,4]$

(ii) $[(\prime a\prime, 1), (\prime a\prime, 2), (\prime a\prime, 3)]$

(iii) $Rectangle\ 6.0\ 14.0$

Question 2

In Haskell, the type constructor *Tree* defined by

$$data\ Tree\ a = Leaf\ a \mid Node\ a\ (Tree\ a)\ (Tree\ a)$$

can be used to represent binary trees with data at internal nodes and leaves.

a) Define a function

$$sumLeaves :: Tree\ Integer\ {-}{>}\ Integer$$

which adds-up the data values just at the leaves

(3 marks)

b) Define a function

$$sumNodes :: Tree\ Integer\ {-}{>}\ Integer$$

which adds-up the data values just at the internal (non-leaf) nodes

(3 marks)

c) Define a function

$$sumTree :: Tree\ Integer\ {-}{>}\ Integer$$

which adds-up the data values at all nodes in a tree. Do not use recursion in your definition.

(4 marks)

Answer:

a) sumLeaves (Leaf n) = n
sumLeaves (Node n t1 t2) = sumLeaves t1 + sumLeaves t2

b) sumNodes (Leaf n) = 0
sumNodes (Node n t1 t2) = n + sumNodes t1 + sumNodes t2

c) sumTree t = sumLeaves t + sumNodes t

Question 3

a) Prove that, for any finite list xs,

$$xs ++ [\,] = [\,] ++ xs$$

(5 marks)

b) Prove that, for any finite lists xs and ys,

$$length\ (xs ++ ys) = length\ xs\ \ + length\ ys$$

(5 marks)

c) Prove for finite lists that

$$sumList\ .\ map\ (2\ *) = (2\ *)\ .\ sumList$$

where

$$\begin{aligned} &sumList :: [Integer] -> Integer \\ &sumList\ [\,] \qquad\quad = 0 \\ &sumList\ (x : xs) = x + sumList\ xs \end{aligned}$$

(10 marks)

Answer:

a) Base case:
[ ] ++ [ ] = [ ] ++ [ ] (equality is reflexive)

Assume that for any list xs, xs ++ [ ] = [ ] ++ xs.

To prove: for any list xs and any x, (x:xs) ++ [ ] = [ ] ++ (x:xs)

$$\begin{aligned} &(x : xs) ++[\,] && \text{((by definition of ++, second equation, left to right))} \\ &= x : (xs ++[\,]) && \text{((by assumption))} \\ &= x : ([\,] ++xs) && \text{((by definition of ++, first equation, left to right))} \\ &= x : xs && \text{((by definition of ++, first equation, right to left))} \\ &= [\,] ++(x : xs) \end{aligned}$$

as required.

b) Base case: for any list ys,
length ([ ] ++ ys) = length [ ] + length ys

$$\begin{aligned} &length([\,] ++ys) && \text{((definition of ++, first equation, left to right))} \\ &= length\ ys && \text{((arithmetic))} \\ &= 0 + length\ ys && \text{((definition of length, first equation, right to left))} \\ &= length[\,] + length\ ys \end{aligned}$$

as required.

Assume for any lists xs and ys, length (xs ++ ys) = length xs + length ys

To prove: for any value x and any lists xs and ys,
length ((x:xs) ++ ys) = length (x : xs) + length ys

$$\begin{array}{lr} length((x:xs)++ys) & \text{((definition of ++, second equation, l to r))} \\ = length(x:(xs++ys)) & \text{((definition of length, sec. equation, l to r))} \\ = 1 + length(xs++ys) & \text{((by assumption))} \\ = 1 + length\ xs + length\ ys & \text{((definition of length, sec. eqn., r to l))} \\ = length(x:xs) + length\ ys & \end{array}$$

as required.

c) *From the type of the expressions we see that we have to show, for any list xs, sumList . map (2*) xs = (2*) . sumList xs*

Base case:
sumList . map (2) [ ] = (2*) . sumList [ ]*

$$\begin{array}{lr} sumList.map(2*)[\,] & \text{((definition of .))} \\ = sumList(map(2+)[\,]) & \text{((definition of map, first equation, l to r))} \\ = sumList[\,] & \text{((definition of sumList, first equation, left to right))} \\ = 0 & \text{((arithmetic))} \\ = (2*)0 & \text{((definition of sumList, first equation, right to left))} \\ = (2*)(sumList[\,]) & \text{((definition of .))} \\ = (2*).sumList[\,] & \end{array}$$

as required.

Assume that for all xs, sumList . map (2) xs = (2*) . sumList xs*

To prove: for any value x and all lists xs, sumList . map (2) (x:xs) = (2*) . sumList (x:xs)*

$$\begin{array}{lr} sumList.map(2*)(x:xs) & \text{((definition of .))} \\ = sumList(map(2*)(x:xs)) & \text{((definition of map, sec. eqn, l to r)} \\ = sumList((2*)x : map(2*)xs)) & \text{((def. of sumList, sec. eqn., l to r))} \\ = (2*)x + sumList(map(2*)xs) & \text{((definition of .))} \\ = (2*)x + sumList.map(2*)xs & \text{((assumption))} \\ = (2*)x + (2*).sumListxs & \text{((definition of .))} \\ = (2*)x + (2*)sumListxs & \text{((arithmetic))} \\ = (2*)(x + sumListxs) & \text{((definition of sumList, sec. eqn., r to l))} \\ = (2*)sumList(x:xs) & \text{((definition of .))} \\ = (2*).sumList(x:xs) & \end{array}$$

as required.

Question 4

a) In the Appendix is the code, using the Parser module you used for your coursework, for parsing the expression part (given by the non-terminal *exp*) from the following grammar for the programming language TINY (also used in your coursework):

$$
\begin{array}{lcl}
cmd & ::= & comp\ ;\ cmd \mid comp \\
comp & ::= & ide := exp \mid output\ exp \mid \\
 & & if\ exp\ then\ cmd\ else\ cmd\ fi \mid \\
 & & while\ exp\ do\ cmd \mid (cmd) \\
exp & ::= & term + exp \mid term = exp \mid term \\
term & ::= & not\ exp \\
factor & ::= & read \mid false \mid true \mid 0 \mid 1 \mid ide \mid (exp) \\
ide & ::= & a\ string\ of\ characters
\end{array}
$$

Write compatible Haskell code to complete the parser for TINY, i.e. write the code to deal with the non-terminals *cmd* and *comp*.

The type of commands that should be the target for the parser is given by

$$
\begin{array}{lcl}
data\ Cmd & = & Assign\ Ide\ Exp \mid Output\ Exp \mid \\
 & & IfThenElse\ Exp\ Cmd\ Cmd \mid \\
 & & WhileDo\ Exp\ Cmd \mid Seq\ Cmd\ Cmd \\
 & & deriving\ Show \\
 & & \\
data\ Exp & = & Plus\ Exp\ Exp \mid Equal\ Exp\ Exp \mid \\
 & & Not\ Exp \mid Read \mid FF \mid TT \mid Zero \mid One \mid I\ Ide \\
 & & deriving\ Show \\
 & & \\
type\ Ide & = & String
\end{array}
$$

(10 marks)

Answer:

```
cmd :: Parser Cmd
cmd  =  do c1 <- comp
           do symbol "; "
              c2 <- cmd
              return (Seq c1 c2)
        +++
        comp

comp :: Parser Cmd
comp = do i <- identifier
          do symbol " :="
             e <- expr
             return (Assign i e)
       +++
       do symbol "output"
          e <- expr
          return(Output e)
       +++
       do symbol "if"
          e <- expr
          do symbol "then"
             c1 <- cmd
             do symbol "else"
                c2 <- cmd
                return (IfThenElse e c1 c2)
       +++
       do symbol "while"
          e <- expr
          do symbol "do"
             c <- cmd
             return (WhileDo e c)
       +++
       do symbol "("
          c <- cmd
          symbol ")"
          return c
```

b) Add productions for declarations of variables, procedures (a named command) and functions (a named expression) to the grammar. Use the non-terminal symbol *decl* to stand for these declarations.

Examples of declarations that your grammar extension should allow for are:

$var\ x = 2$
$var\ sum = x + 4$
$var\ x = 2;\ var\ sum = 4$
$proc\ f(x);\ (var\ y = 0;\ output\ (x + y))$
$fun\ decr(n);\ n - 1$

where $x, sum, f, y, decr$ and n are all examples of identifiers.

(10 marks)

Answer:

$$
\begin{array}{lcl}
decl & ::= & decls;\ decl \mid decls \\
decls & ::= & var\ ide = exp \mid proc\ ide\ (ide);\ cmd \mid fun\ ide\ (ide);\ exp
\end{array}
$$

Note: *as ever, the stratification is important here, so take of some (not all) marks for not doing it.*

c) (i) Write code for a function *decl* which parses your new declaration productions. Use the data structure given by:

$$data\ Decl = Var\ Ide\ Exp \mid Proc\ Ide\ Ide\ Cmd \mid Fun\ Ide\ Ide\ Exp$$

as the target of your new piece of parsing code.

(5 marks)

Answer:

```
decl :: Parser Decl
decl = do d1  <- decls
          symbol "; "
          d2 <- decl
          return (Seqd d1 d2)
       +++
       decls
decls:: Parser Decl
decls= do symbol "var"
          i <- identifier
          symbol " ="
          e  <- expr
          return (Var i e)
       +++
       do symbol "proc"
          i1 <- identifier
         symbol "("
       i2  <- identifier
          symbol ")"
          symbol "; "
          c  <- cmd
          return (Proc i1 i2 c)
       +++
       do symbol "fun"
          i1 <- identifier
         symbol "("
       i2  <- identifier
          symbol ")"
          symbol "; "
          e  <- expr
          return (Fun i1 i2 e)
```

Note: *there is a bit of a sting here—they should have pointed out that the data type needed an extra clause and then stratification as usual in the code, all to deal with sequences of declarations.*

(ii) In order to include declarations within commands we need to add a new production for commands:

$$cmd ::= begin\ decl\ ;\ cmd\ end$$

Show how to extend the code of the parser for commands to add this production to the parser.

(5 marks)

Answer:
Need to add this alternative to the function comp that parses commands:

```
+++
do symbol "begin"
   d <- decl
   symbol "; "
   c <- cmd
   symbol "end"
   return (BeginEnd d c)
```

and also need to add a clause to the data type for commands to introduce the new constructor BeginEnd:

BeginEnd Decl Cmd

Question 5

a) Using the semantic clauses for TINY given in the Appendix, evaluate:

(i)

$$C[\![output\ 1;\ output\ 0]\!](\emptyset, <>, <>)$$

(5 marks)

Answer:
First two partial results:
(a)

$$\begin{aligned}
& C[\![output\ 1]\!](\emptyset, <>, <>) \\
&= (E[\![output\ 1]\!](\emptyset, <>, <>) = (v, (m, i, o))) \rightarrow (m, i, v.o), error && \text{(C1)} \\
&= (1, (\emptyset, <>, <>)) = (v, (m, i, o)) \rightarrow (m, i, v.o), error && \text{(E1)} \\
&= (\emptyset, <>, < 1 >) && \text{(pattern matching and definition of } \rightarrow)
\end{aligned}$$

(b)

$$\begin{aligned}
& C[\![output\ 0]\!](\emptyset, <>, < 1 >) \\
&= (E[\![output\ 0]\!](\emptyset, <>, < 1 >) = (v, (m, i, o))) \rightarrow (m, i, v.o), error && \text{(C1)} \\
&= (0, (\emptyset, <>, < 1 >)) = (v, (m, i, o)) \rightarrow (m, i, v.o), error && \text{(E1)} \\
&= (\emptyset, <>, < 0.1 >) && \text{(pattern matching and definition of } \rightarrow)
\end{aligned}$$

So, main result:

$$\begin{aligned}
& C[\![output\ 1;\ output\ 0]\!](\emptyset, <>, <>) \\
&= (C[\![output\ 1]\!](\emptyset, <>, <>) = error) \rightarrow \\
&\qquad error, C[\![output\ 0]\!](C[\![output\ 1]\!](\emptyset, <>, <>)) && \text{(C5)} \\
&= C[\![output\ 0]\!](\emptyset, <>, < 1 >) \ \ \text{(By (a) and } (\emptyset, <>, < 1 >) \neq error) \\
&= (\emptyset, <>, < 0.1 >) && \text{(By (b))}
\end{aligned}$$

(ii)

$$C[\![output(read + read)]\!](\emptyset, < 1, 2 >, <>)$$

(5 marks)

Answer:
First two partial results:

(a)

$$
\begin{aligned}
& E[\![read + read]\!](\emptyset, <1,2>, <>) \\
&= (E[\![read]\!](\emptyset, <1,2>, <>) = (v_1, s_1)) \rightarrow \\
&\qquad ((E[\![read]\!]s_1 = (v_2, s_2)) \rightarrow \\
&\qquad\qquad (isNum\ v_1\ and\ isNum\ v_2 \rightarrow \\
&\qquad\qquad\qquad (v_1 + v_2, s_2), error), error), error \qquad (\text{E7}) \\
&= (1, (\emptyset, <2>, <>) = (v_1, s_1)) \rightarrow \\
&\qquad ((E[\![read]\!]s_1 = (v_2, s_2)) \rightarrow \\
&\qquad\qquad (isNum\ v_1\ and\ isNum\ v_2 \rightarrow \\
&\qquad\qquad\qquad (v_1 + v_2, s_2), error), error), error \\
&\qquad (\text{E3 and } null\ <1,2> = false \text{ and } hd <1,2> = 1 \text{ and } tl <1,2> = <2>) \\
&= ((E[\![read]\!](\emptyset, <2>, <>) = (v_2, s_2)) \rightarrow \\
&\qquad\qquad (isNum\ 1\ and\ isNum\ v_2 \rightarrow \\
&\qquad\qquad\qquad (1 + v_2, s_2), error), error) \\
&\qquad (\text{Pattern matching and definition of } \rightarrow) \\
&= (2, (\emptyset, <>, <>)) = (v_2, s_2) \rightarrow \\
&\qquad (isNum\ 1\ and\ isNum\ v_2 \rightarrow \\
&\qquad\qquad (1 + v_2, s_2), error), error) \\
&\qquad (\text{E3 and } null\ <2> = false \text{ and } hd <2> = 2 \text{ and } tl <2> = <>) \\
&= isNum\ 1\ and\ isNum\ 2 \rightarrow (1 + 2, (\emptyset, <>, <>)), error \\
&\qquad (\text{Pattern matching and definition of } \rightarrow) \\
&= (1 + 2, (\emptyset, <>, <>)) \\
&\qquad (isNum\ 1\ and\ isNum\ 2 = true \text{ and definition of } rightarrow) \\
&= (3, (\emptyset, <>, <>)) \qquad (\text{arithmetic})
\end{aligned}
$$

Then the main result:

$$
\begin{aligned}
& C[\![output(read + read)]\!](\emptyset, <1,2>, <>) \\
&= (\emptyset, <>, <3>) \qquad (\text{C1, (a), pattern matching, definition of } \rightarrow)
\end{aligned}
$$

where $\emptyset$ is the function which has empty domain and range, $<1,2>$ is the sequence consisting of 1 followed by 2 and $<>$ is the empty sequence.

b) Using the semantic clauses for SMALL given in the Appendix (photocopied from chapter six of Gordon's book), evaluate:

(i)

$$P[\![program\ output\ 1;\ output\ 0\ ]\!] <>$$

(5 marks)

Answer:
In these calculations I have put almost every little detail in. I do not expect the students to have spelt out each step in this much detail, but each step taken (which might bundle together several of the steps I give here) should be fairly well-justified.

$P[\![program\ \ output\ 1;\ output\ 0]\!] <>$

$= C[\![output\ 1;\ \ output\ 0]\!](\ )\ (\lambda\, s.stop)(<> /input)$ (P)

$= C[\![output\ 1]\!](\ )\ ;\ \ C[\![output\ 0]\!](\ )\ ;\ (\lambda\, s.stop)(<> /input)$ (C7)

$= C[\![output\ 1]\!](\ )\ (C[\![output\ 0]\!](\ )\ (\lambda\, s.stop))(<> /input)$
(definition of ;)

$= R[\![1]\!](\ )\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(<> /input)$
(C2)

$= E[\![1]\!](\ )\ ;\ \ deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(<> /input)$
(R)

$= E[\![1]\!](\ )\ (deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s))(<> /input)$
(definition of ;)

$= deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(B[\![1]\!])(<> /input)$
(E1)

$= deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(1)(<> /input)$
$(B[\![1]\!] = 1)$

$= deref(Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s))(1)(<> /input)$
(definition of ;)

$= isloc\ 1 \rightarrow cont\ (Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s))(1)(<> /input),$
$Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(1)(<> /input)$
(definition of deref)

$= Rv?\ ;\ \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(1)(<> /input)$
(definition of isLoc)

$= Rv?(\lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s))(1)(<> /input)$
(definition of ;)

$= (isRv\ 1 \rightarrow \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s))(1), err)(<> /input)$
(definition of Rv?)

$= \lambda\ \ e\ s.(e,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ s)(1)(<> /input)$
(definition of isRv)

$= (1,\ (C[\![output\ 0]\!](\ )(\lambda\, s.stop))\ (<> /input))$ (application)

$= (1,\ R[\![0]\!](\ )\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(<> /input))$ (C2)

$= (1,\ E[\![0]\!](\ )\ ;\ \ deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(<> /input))$
(R)

$= (1,\ E[\![0]\!](\ )\ (deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s))(<> /input))$
(definition of ;)

$= (1,\ deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(B[\![0]\!])(<> /input))$
(E1)

$= (1,\ deref;\ \ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(0)(<> /input))$
$(B[\![0]\!] = 0)$

$= (1,\ deref(Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s))(0)(<> /input))$
(definition of ;)

$= (1,\ isloc\ 0 \to cont\ (Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s))(0)(<>/input),$
$Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(0)(<>/input))$ (definition of deref)

$= (1,\ Rv?\ ;\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(0)(<>/input))$ (definition of isLoc)

$= (1,\ Rv?(\lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s))(0)(<>/input))$ (definition of ;)

$= (1,\ (isRv\ 0 \to \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(0), err)(<>/input))$ (definition of Rv?)

$= (1,\ \lambda\ \ e\ s.(e,\ (\lambda\, s.stop)\ s)(0)(<>/input))$ (definition of isRv)

$= (1,\ (0, \lambda\, s.stop\ (<>/input)))$ (application)

$= (1,\ (0, stop))$ (application)

(ii)

$$P\llbracket program\ begin\ var\ x = read;\ output\ x\ end\rrbracket <>$$

(5 marks)

Answer:

$P\llbracket program\ begin\ var\ x\ =\ read;\ output\ x\ end\rrbracket <>$

$= C\llbracket begin\ var\ x\ =\ read;\ output\ x\ end\rrbracket(\ )\ (\lambda\, s.stop)(<>/input)$ (P)

$= D\llbracket var\ x\ =\ read\rrbracket(\ )\,\lambda\, r'.C\llbracket output\ x\rrbracket(\ )[r']\ (\lambda\, s.stop)(<>/input)$ (C6)

$= R\llbracket read\rrbracket(\ );\ ref\ \ \lambda\,\iota.\,\lambda\, r'.C\llbracket output\ x\rrbracket(\ )[r'](\iota/x)\ (\lambda\, s.stop)(<>/input)$ (D2)

$= R\llbracket read\rrbracket(\ )(ref\ \ \lambda\,\iota.\,\lambda\, r'.C\llbracket output\ x\rrbracket(\ )[r'](\iota/x)\ (\lambda\, s.stop))(<>/input)$ (definition of ;)

$= E\llbracket read\rrbracket(\ );\ deref;\ Rv?;\ (ref\ \ \lambda\,\iota.\,\lambda\, r'.C\llbracket output\ x\rrbracket(\ )[r'](\iota/x)\ (\lambda\, s.stop))(<>/input)$ (R)

$= E\llbracket read\rrbracket(\ )\ (deref;\ Rv?;\ (ref\ \ \lambda\,\iota.\,\lambda\, r'.C\llbracket output\ x\rrbracket(\ )[r'](\iota/x)\ (\lambda\, s.stop)))(<>/input$ (definition of ;)

$= null((<>/input)\ input) \to error, ...$ (E3)

$= null(\ ) \to error, ...$ (((¡¿/input)) input = ¡¿)

$= error$ (null¡¿ = true)

You must show all your working in detail.

Question 6

In TINY we can define a new command *donothing*, which has no effect on the state, by:

$$C[\![donothing]\!]\ s = s$$

Given this new command, show that:

$$C[\![C;\ donothing]\!]\ s = C[\![donothing;\ C]\!]\ s = C[\![C]\!]\ s$$

for all commands C and states s.

You must show all your working in detail.

(10 marks)

Answer:

$$\begin{aligned}
&C[\![C;\ donothing]\!]\ s \\
&= (C[\![C]\!]\ s = error) \rightarrow error, C[\![donothing]\!]\ (C[\![C]\!]\ s) && \text{(C5)} \\
&= (C[\![C]\!]\ s = error) \rightarrow error, C[\![C]\!]\ s && \text{(by definition of } donothing\text{)} \\
&= C[\![C]\!]\ s \\
&\qquad \text{(consider the cases where } C[\![C]\!]\ s = error \text{ and } C[\![C]\!]\ s \neq error\text{)}
\end{aligned}$$

$$\begin{aligned}
&C[\![donothing;\ C]\!]\ s \\
&= (C[\![donothing]\!]\ s = error) \rightarrow error, C[\![C]\!]\ (C[\![donothing]\!]\ s) && \text{(C5)} \\
&= C[\![C]\!]\ s && \text{(by definition of } donothing\text{)}
\end{aligned}$$

Appendix

Definitions of various Haskell functions

$$\begin{aligned}
&map :: (a \ {-}{>}\ b) \ {-}{>}\ [a] \ {-}{>}\ [b] \\
&map\ f\ [\,] \qquad = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}$$

$$\begin{aligned}
&(++) :: [a] \ {-}{>}\ [a] \ {-}{>}\ [a] \\
&[\,] ++\ ys \qquad = ys \\
&(x : xs)\ ++\ ys = x : (xs\ ++\ ys)
\end{aligned}$$

$$\begin{aligned}
&length :: [a] \ {-}{>}\ Integer \\
&length\ [\,] \qquad = 0 \\
&length\ (x : xs) = 1 + length\ xs
\end{aligned}$$

Haskell for parser

Here is the code for the expression parts of the parser for TINY:

```
exp  :: Parser Exp
exp  = do e1 <- term
          do symbol "+"
             e2 <- exp
             return(Plus e1 e2)
       +++
       do e1 <- term
          do symbol "="
             e2 <- exp
             return (Equal e1 e2)
       +++
       term

term :: Parser Exp
term = do symbol "not"
          e <- exp
          return (Not e)
       +++
       factor

factor:: Parser Exp
factor= do symbol "read"
           return Read
        +++
        do symbol "false"
           return FF
        +++
        do symbol "true"
           return TT
        +++
        do symbol "0"
           return Zero
        +++
        do symbol "1"
           return One
        +++
        do i <- identifier
           return (I i)
        +++
        do symbol "("
           e <- exp
           do symbol ")"
              return e
```

Semantic clauses for TINY

First the semantic domains:

$$\begin{array}{lcl} \mathit{State} & = & \mathit{Memory} \times \mathit{Input} \times \mathit{Output} \\ \mathit{Memory} & = & \mathit{Ide} \to [\mathit{Value} + \{\mathit{unbound}\}] \\ \mathit{Input} & = & \mathit{Value}^* \\ \mathit{Output} & = & \mathit{Value}^* \\ \mathit{Value} & = & \mathit{Num} + \mathit{Bool} \end{array}$$

where *Ide* is a domain of identifiers, and *Num* and *Bool* are basic values that can be represented in the language.

Next the clauses for expressions:

$$E : \mathit{Exp} \to [\mathit{State} \to [\mathit{Value} + \{\mathit{error}\}]]$$

where *Exp* is the syntactic domain of expressions.

E1

$$\begin{array}{l} E[\![0]\!]\ s = (0, s) \\ E[\![1]\!]\ s = (1, s) \end{array}$$

E2

$$\begin{array}{l} E[\![\mathit{true}]\!]\ s = (\mathit{true}, s) \\ E[\![\mathit{false}]\!]\ s = (\mathit{false}, s) \end{array}$$

E3

$$E[\![\mathit{read}]\!]\ (m, i, o) = \mathit{null}\ i \to \mathit{error}, (\mathit{hd}\ i, (m, \mathit{tl}\ i, o))$$

E4

$$E[\![I]\!]\ (m, i, o) = (m\ I = \mathit{unbound}) \to \mathit{error}, (m\ I, (m, i, o))$$

E5

$$E[\![\mathit{not}\ E]\!]\ s = (E[\![E]\!]\ s = (v, s')) \to (\mathit{isBool}\ v \to (\mathit{not}\ v, s'), \mathit{error}), \mathit{error}$$

E6

$$\begin{array}{l} E[\![E_1 = E_2]\!]\ s = (E[\![E_1]\!]\ s = (v_1, s_1)) \to \\ \qquad ((E[\![E_2]\!]\ s_1 = (v_2, s_2)) \to (v_1 = v_2, s_2), \mathit{error}), \mathit{error} \end{array}$$

E7

$$\begin{array}{l} E[\![E_1 + E_2]\!]\ s = (E[\![E_1]\!]\ s = (v_1, s_1)) \to \\ \qquad ((E[\![E_2]\!]\ s_1 = (v_2, s_2)) \to \\ \qquad\quad (\mathit{isNum}\ v_1\ \mathit{and}\ \mathit{isNum}\ v_2 \to \\ \qquad\qquad (v_1 + v_2, s_2), \mathit{error}), \mathit{error}), \mathit{error} \end{array}$$

Now the clauses for commands:

$$C : Com \rightarrow [State \rightarrow [State + \{error\}]]$$

where *Com* is the syntactic domain of commands.

C1

$$C[\![output\ E]\!]\ s = (E[\![E]\!]\ s = (v, (m, i, o))) \rightarrow (m, i, v.o), error$$

C2

$$C[\![I :=\ E]\!]\ s = (E[\![E]\!]\ s = (v, (m, i, o))) \rightarrow (m[v/I], i, o), error$$

C3

$$\begin{aligned} C[\![if\ E\ then\ C_1\ else\ C_2]\!]\ s = (E[\![E]\!]\ s = (v, s')) \rightarrow & \\ & (isBool\ v \rightarrow \\ & \quad (v \rightarrow C[\![C_1]\!]\ s', C[\![C_2]\!]\ s'), error), error \end{aligned}$$

C4

$$\begin{aligned} C[\![while\ E\ do\ C]\!]\ s = ((E[\![E]\!]\ s = (v, s')) \rightarrow & \\ & (isBool\ v \rightarrow (v \rightarrow \\ & \quad ((C[\![C]\!]\ s' = s'') \rightarrow C[\![while\ E\ do\ C]\!]\ s'', error) \\ & \qquad , s'), error), error) \end{aligned}$$

C5

$$C[\![C_1;\ C_2]\!]\ s = (C[\![C_1]\!]\ s = error) \rightarrow error, C[\![C_2]\!]\ (C[\![C_1]\!]\ s)$$

Semantic clauses for SMALL

These are attached overleaf.