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. [Three pages are provided, but are not printed here]

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) $map\ (\backslash\ x{-}{>}\ ('a', x))[1, 2, 3]$

(6 marks)

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

(i) $map\ (1+)\ [1, 2, 3]$
(ii) $(map\ (\backslash\ x{-}{>}\ ('a', x))\ [1, 2, 3]$
(iii) $f\ (Rectangle\ 6.0\ 7.0)\ 1.0\ 2.0$
$where\ f\ s\ dx\ dy = case\ s\ of$
$RtTriangle\ s1\ s2 = RtTriangle\ s1 * dx\ s2 * dy$
$Rectangle\ s1\ s2 = Rectangle\ s1 * dx\ s2 * dy$
$Ellipse\ r1\ r2 = Ellipse\ r1 * dx\ r2 * dy$

given the data declaration

$$data\ Shape = Rectangle\ Float\ Float\ |\ Ellipse\ Float\ Float\ |\ RtTriangle\ Float\ Float$$

(4 marks)

2. In Haskell, the type constructor *Tree* defined by

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

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)

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

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

(10 marks)

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):

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

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

```
data Cmd = Assign Ide Exp | Output Exp |
           IfThenElse Exp Cmd Cmd |
           WhileDo Exp Cmd | Seq Cmd Cmd
           deriving Show

data Exp = Plus Exp Exp | Equal Exp Exp |
           Not Exp | Read | FF | TT | Zero | One | I Ide
           deriving Show

type Ide = String
```

(10 marks)

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:

$$
\begin{array}{l}
\mathit{var}\ x = 2 \\
\mathit{var}\ \mathit{sum} = x + 4 \\
\mathit{var}\ x = 2;\ \mathit{var}\ \mathit{sum} = 4 \\
\mathit{proc}\ f(x);\ (\mathit{var}\ y = 0;\ \mathit{output}\ (x + y)) \\
\mathit{fun}\ \mathit{decr}(n);\ n - 1
\end{array}
$$

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

(10 marks)

c) (i) Write code for a function *decl* which parses your new declaration productions.

Use the data structure given by:

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

as the target of your new piece of parsing code.

(5 marks)

Question 4 continued on next page

(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)

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

(i)

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

(5 marks)

(ii)

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

(5 marks)

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)

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

(5 marks)

(ii)

$$P[\![program\ begin\ var\ x = read;\ output\ x\ end]\!] <>$$

(5 marks)

You must show all your working in detail.

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)

Appendix

Definitions of various Haskell functions

```
map :: (a -> b) -> [a] -> [b]
map f [ ]       = [ ]
map f (x : xs) = f x : map f xs
```

```
(++) :: [a] -> [a] -> [a]
[ ] ++ ys       = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

```
length :: [a] -> Integer
length [ ]       = 0
length (x : xs) = 1 + length xs
```

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(Pluse1e2)
      +++
      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

$$E[\![0]\!]\ s = (0, s)$$
$$E[\![1]\!]\ s = (1, s)$$

E2

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

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

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

E7

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

CONTINUED

Now the clauses for commands:

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

where *Com* is the syntactic domain of commands.

C1

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

C2

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

C3

$$C[\![if\ E\ then\ C_1\ else\ C_2]\!]\ s = (E[\![E]\!]\ s = (v, s')) \to (isBool\ v \to (v \to C[\![C_1]\!]\ s', C[\![C_2]\!]\ s'), error), error$$

C4

$$C[\![while\ E\ do\ C]\!]\ s = ((E[\![E]\!]\ s = (v, s')) \to (isBool\ v \to (v \to ((C[\![C]\!]\ s' = s'') \to C[\![while\ E\ do\ C]\!]\ s'', error), s'), error), error)$$

C5

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

Semantic clauses for SMALL

These are attached overleaf.