COMPSCI 3MI3

Assignment 2

1 Solution Set

1.1 Q1

The following EBNF grammar describes the song "Hey Jude" as given by the diagram.

$\langle song \rangle$ ::= $\langle verses \rangle$ {$\langle verses \rangle$}

$\langle verses \rangle$ ::= '`hey Jude`' $\langle v1 \rangle$ $\langle v2 \rangle$ $\langle v3 \rangle$ $\langle v4 \rangle$ $\langle v5 \rangle$

$\langle v1 \rangle$ ::= '`don't`' $\langle c1 \rangle$

$\langle c1 \rangle$ ::= '`make it bad take a sad song and make it better`'
| '`be afraid you were made to go out and get her`'
| '`let me down you have found here, now go and get her`'

$\langle v2 \rangle$ ::= '`remember to`' $\langle c2 \rangle$

$\langle c2 \rangle$ ::= '`let her into your heart`' | '`let her under your skin`'

$\langle v3 \rangle$ ::= '`then you`' ('`can start`' | '`begin`') '`to make it better`'

$\langle v4 \rangle$ ::= '`better better better better better waaaaa`'

$\langle v5 \rangle$ ::= '`na`' {$\langle v5 \rangle$}

1.2 Q2

- The following EBNF grammar describes identifiers in Python.

  $\langle identifier \rangle$ ::= $\langle underscore \rangle${$\langle char \rangle$} | $\langle letter \rangle${$\langle char \rangle$}

⟨*char*⟩ ::= ⟨*digit*⟩ | ⟨*letter*⟩ | ⟨*underscore*⟩

⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*letter*⟩ ::= ⟨*capital*⟩ | a | b | c | d | e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x | y | z |

⟨*capital*⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨*underscore*⟩ ::= ‘_’

- The following EBNF grammar describes identifiers in Haskell.

⟨*identifier*⟩ ::= ⟨*letter*⟩{⟨*char*⟩} | ⟨*underscore*⟩⟨*char*⟩{⟨*char*⟩}

⟨*char*⟩ ::= ⟨*digit*⟩ | ⟨*letter*⟩ | ⟨*underscore*⟩ | ⟨*single-quote*⟩

⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*letter*⟩ ::= ⟨*capital*⟩ | a | b | c | d | e | f | g | h | i | j | k
| l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

⟨*capital*⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z

⟨*underscore*⟩ ::= ‘_’

⟨*single-quote*⟩ ::= ‘’’

- The following EBNF grammar describes for loops in C.

  Assume that statements are generated by the rule for `<statement>` and expressions by `<expression>`.

⟨*loop*⟩ ::= for ⟨*condition*⟩ ‘{’ ⟨*body*⟩ ‘}’

⟨*condition*⟩ ::= ‘`(`’[⟨*statement*⟩]‘`;`’ [⟨*expression*⟩]‘`;`’ [⟨*statement*⟩]‘`)`’

⟨*body*⟩ ::= { ⟨*statement*⟩‘`;`’ }

1.3 Q3

We define the following Python terms using inference rule style definitions shown in lecture. For all definitions assume $\mathcal{T}$ to be the set of terms in Python.

- for loop

$$\frac{x \in \mathcal{T} \qquad \mathit{iter} \in \mathcal{T} \qquad b_1 \in \mathcal{T} \qquad b_2 \in \mathcal{T}}{\texttt{for}\, x\, \texttt{in}\, \mathit{iter}\texttt{:\textbackslash n\textbackslash t}\, b_1\, \texttt{else:\textbackslash n\textbackslash t}\, b_2 \in \mathcal{T}}$$

- while loop

$$\frac{\mathit{condition} \in \mathcal{T} \qquad b_1 \in \mathcal{T} \qquad b_2 \in \mathcal{T}}{\texttt{while}\, \mathit{condition}\texttt{:\textbackslash n\textbackslash t}\, b_1\, \texttt{else:\textbackslash n\textbackslash t}\, b_2 \in \mathcal{T}}$$

- assignment statement

$$\frac{\mathit{identifier} \in \mathcal{T} \qquad \mathit{expression} \in \mathcal{T}}{\mathit{identifier}\, \texttt{=}\, \mathit{expression} \in \mathcal{T}}$$

- slicing operation

$$\frac{x \in \mathcal{T} \qquad \mathit{start} \in \mathcal{T} \qquad \mathit{end} \in \mathcal{T} \qquad \mathit{step} \in \mathcal{T}}{x\texttt{[}\mathit{start}\texttt{:}\mathit{end}\texttt{:}\mathit{step}\texttt{]}}$$

- function declaration

$$\frac{\mathit{name} \in \mathcal{T} \qquad \mathit{args} \in \mathcal{T} \qquad \mathit{block} \in \mathcal{T}}{\texttt{def}\, \mathit{name}\texttt{(}\, \mathit{args}\, \texttt{):\textbackslash n\textbackslash t}\, \mathit{block} \in \mathcal{T}}$$

1.4 Q4

(a) See A2_Q4.lhs for code.

We define the syntax for Untyped Arithemtic Expressions in Haskell as follows:

```
module A2 where

data UAETerm =
      T     -- True
    | F     -- False
    | Zero
    | IfThenElse UAETerm UAETerm UAETerm
    | Succ UAETerm
    | Pred UAETerm
    | IsZero UAETerm
    deriving (Eq, Show)
```

(b) See A2_Q4.lhs for code.

We define the following unary Haskell functions to encode the given arithmetic expressions.

```
-- (1)
exp1 :: UAETerm
exp1 = Succ $ Succ $ Succ Zero

-- (2)
exp2 :: UAETerm
exp2 = Succ $ Pred $ IsZero $ Succ T

-- (3)
exp3 :: UAETerm
exp3 = IfThenElse x y z
    where
        x = IsZero $ Succ $ Pred Zero
        y = Pred Zero
        z = Succ $ Succ $ Succ Zero
```

(c) There are semantic errors in some of the expressions from (b).

(1) No semantic errors.

(2) The syntactic expression succ true does not make sense semantically since succ semantically means the successor of a natural number. Thus the successor of a truth value does not make sense semantically.

Also, the sub-expression ... pred (iszero (... also does not make sense semantically. Semantically we expect iszero to evaluate to a truth value, checking whether its argument is equal to zero. However pred semantically evaluates to the predecessor of a natural number. Therefore it does not makes sense to apply the predecessor to a truth value.

(3) The syntactic sub-expression ... succ (pred 0))) ... does not makes sense semantically if we assume the semantics of the natural numbers as given in the slides. We cannot take the predecessor of zero since it is the lowest natural number.

The sub-expression (pred false) in the then section of the expression does not make sense semantically since pred semantically should evaluate to the predecessor of a natural number, and thus should not applied to a truth value like false.

Overall the if-expression could have semantic errors depending on the evaluated type of (pred false) in the then expression. If it evaluates to a boolean value the entire expression could evalute to either a truth value *or* a numerical value depending if the predicate is true or false semantically. Semantically, within a type system the expression should always evaluate to the same type.