

INFOB3TC – Solutions for Exam 1

Johan Jeuring

Wednesday, 16 December 2015, 08:30–10:30

Please keep in mind that there are often many possible solutions and that these example solutions may contain mistakes.

Questions

NinjaPoke studios sells a Unity asset called 'Dialogue'¹, for five dollars.



The Dialogue asset is used for implementing dialogues in games. In this series of exercises we will create part of the functionality of a slightly extended version of this asset.

Here is a simple example dialogue between a player and a friend (presumably the player represent the person playing the dialogue, and the friend is a virtual character, but this need not be the case):

```
Player: Hello!  
Friend: Hi!  
Player: How are you?  
Friend: I'm good
```

¹<https://www.assetstore.unity3d.com/>

A dialogue consists of a list of 'stage directions' (as NinjaPoke calls them), where each stage direction consists of an identifier for the character making a statement (in this case Player and Friend, but you may use any names here), followed by a colon, followed by one or more spaces, and then a sentence. There is a newline at the end of such a sentence.

Instead of a sentence, you can also offer a choice between various sentences, so that a player (or anyone else) can choose between different options. Every choice has a score, and can be followed by a different part of the dialogue. For example:

```

Harry: Hello!
Sally: Hi, how are you?
Harry: {I feel tired}      [goto Tired]      2
      {Not bad}           [end]              3
      {I'm feeling sick}  [call BuyMedicine] 1
-Tired
Sally: Go to sleep then

```

The various options offered to Harry (in this case) are surrounded by { and }. The options are followed by a command, which is surrounded by [and], and a score, which is an integer. A command is either end, denoting that a dialogue ends here, a call to a method (an identifier in our case, BuyMedicine), or a jump goto that takes a label (Tired) as argument. Further down in the dialogue the label appears on its own line, preceded by a - (-Tired). Below this label appears the piece of dialogue you want to play if the player selects the option to which this goto label is connected.

1 (12 points). Give a concrete syntax (a context-free grammar) of this language for dialogues. You may use nonterminal *Identifier* to recognise a single name, *String* to recognise the content of a sentence (a string not containing a newline or a closing curly parenthesis `}`), and *Integer* to recognise a score. Describe the language as precisely as possible, but you may ignore occurrences of spaces (you may include them as well). •

Solution 1.

```

Dialogue → Direction*
Direction → Label | Statement
Label → "-" Identifier "\n"
Statement → Character ":" Utterance
Character → Identifier
Utterance → Sentence "\n" | Options
Sentence → String
Options → ("{" Sentence "}" Jump Score)*
Jump → "[" Command "]"
Command → "goto" Identifier | "end" | "call" Method
Method → Identifier
Score → Integer

```

Here is the above example sentence:

```
example = player1 ++ friend1 ++ player2 ++ label ++ friend2
player1 = "Player: Hello!\n"
friend1 = "Friend: Hi, how are you?\n"
player2 = "Player: " ++ option1 ++ option2 ++ option3
option1 = "{Tired} [goto Tired] 2\n"
option2 = "          {Good}  [end] 3\n"
option3 = "          {Sick}  [call BuyMedicine] 1\n"
label   = "-Tired\n"
friend2 = "Friend: Go to sleep then"
```

Marking

2 (12 points). Discuss three grammar transformations: describe in words what they do, give an example of their application, and discuss whether or not they can be applied to the grammar you defined in Task 1, and, if they can be applied, what the result would look like. ●

Solution 2.

Marking

3 (12 points). Define an abstract syntax (a (data) type *Dialogue* in Haskell) that corresponds to your concrete syntax given as an answer in Task 1, which you can use to represent a dialogue in Haskell. ●

Solution 3.

```
type Dialogue = [Direction]
data Direction = Label Label | Statement Statement deriving Show
type Label     = Identifier
type Statement = (Character, Utterance)
type Character = Identifier
data Utterance = Sentence Sentence | Options Options deriving Show
type Sentence = String
type Options  = [(Sentence, Jump, Score)]
type Jump     = Command
data Command = Goto Identifier | End | Call Method deriving Show
type Method  = Identifier
type Score   = Int
type Identifier = String
```

Marking

4 (12 points). Define a parser $pDialogue :: Parser Char Dialogue$ that parses sentences from the language of dialogues. Define your parser using parser combinators. •

Solution 4.

```
pDialogue :: Parser Char Dialogue
pDialogue = many pDirection

pDirection :: Parser Char Direction
pDirection = Label <$> pLabel <|> Statement <$> pStatement

pLabel :: Parser Char Label
pLabel = symbol '- ' *> identifier <*> symbol '\n'

pStatement :: Parser Char Statement
pStatement = (,) <$> (pCharacter <*> tokensp ":" <*>) <*> pUtterance

pUtterance :: Parser Char Utterance
pUtterance = Sentence <$> pSentence <*> symbol '\n' <|> Options <$> pOptions

pOptions :: Parser Char Options
pOptions = many ((,,) <$>
    (tokensp "{" *> pSentence) <*> (tokensp "}" *> pJump) <*> integersp
)

pJump :: Parser Char Jump
pJump = tokensp "[" *> pCommand <*> tokensp "]"

pCommand :: Parser Char Command
pCommand = Goto <$> (tokensp "goto" *> identifier)
    <|> End <$> token "end"
    <|> Call <$> (tokensp "call" *> pMethod)

pMethod, pCharacter, pSentence :: Parser Char Identifier
pMethod = identifier
pCharacter = identifier
pSentence = greedy (satisfy ( $\lambda c \rightarrow (c \neq '\n') \wedge (c \neq '}'')$ ))

spaces = greedy (satisfy isSpace)
tokensp s = token s <*> spaces
integersp = integer <*> spaces

— Parser test case
test = fst $ head $ pDialogue example
```

Marking

Dialogues have a tree-like structure that is not directly visible in the concrete syntax, and hence not in the derived abstract syntax given in Task 3 either. I will now define an abstract syntax that encodes the tree-like structure. To keep things simple for the following exercises, I assume that if there is a choice in a dialogue, there are always exactly two options.

A value of the abstract dialogue data type (*ADialogue*) is either the empty dialogue *ADEnd*, or it is a *Single* statement (a *Sentence* from a particular *Character*) followed by a dialogue, or it is a *Choice* for a *Character* between two options, where each option consists of a *Sentence*, a *Score*, and a dialogue that follows when the *Character* chooses this option.

— Simplified abstract dialogue type in which there are never more than two choices

```

data ADialogue = Single Character Sentence ADialogue
                | Choice Character
                  ( Sentence, ADialogue, Score )
                  ( Sentence, ADialogue, Score )
                | ADEnd
type Character = Identifier
type Sentence = String
type Score    = Int

```

5 (12 points). Define the algebra type, and the *fold* for the data type *ADialogue*. You may assume that the types *Sentence*, *Score*, and *Character* are constant types such as *Int* and *String*, that is, you don't have to define a *fold* for these types. •

Solution 5.

```

type ADialogueAlgebra a = ( Character → Sentence → a → a
                             , Character → ( Sentence, a, Score ) → ( Sentence, a, Score ) → a
                             , a
                             )
foldDialogue :: ADialogueAlgebra a → ADialogue → a
foldDialogue alg@(single, choices, end) = f where
  f ( Single c s a ) = single c s ( f a )
  f ( Choices c ( se1, a1, sc1 ) ( se2, a2, sc2 ) ) = choices c ( se1, f a1, sc1 ) ( se2, f a2, sc2 )
  f ADEnd = end

```

Marking

The following dialogue:

```

Player: Hi
Friend: How are you?
Player: {Good} [goto i0] 3
         {Bad}  [goto i3] 2
-i0
Friend: Good to hear
Player: {Thanks for asking} [end] 2
         {Yes, yes}          [end] 1
-i3
Friend: Sad to hear
Player: {Well, what do you think?} [end] 0
         {Yes, but thanks for asking} [end] 1

```

is represented by the following value of the data type *ADialogue*:

```

exADialogue =
  Single "Player" "Hi"
  $Single "Friend" "How are you?"
  $Choices "Player" ("Good",ead2,3) ("Bad",ead3,2)
ead2 =
  Single "Friend" "Good to hear"
  $Choices "Player" ("Thanks for asking",ADEnd,2) ("Yes, yes",ADEnd,1)
ead3 =
  Single "Friend" "Sad to hear"
  $Choices "Player" ("Well, what do you think?",ADEnd,0)
                  ("Yes, but thanks for asking",ADEnd,1)

```

When a player plays this dialogue, he or she scores points at each choice. The total score of playing a dialogue is the sum of the scores at the choices taken. For example, if the player plays the following dialogue:

```

Player: Hi
Friend: How are you?
Player: Good
Friend: Good to hear
Player: Yes, yes

```

he or she scores 4 points in total.

6 (10 points). Define a function $maxScore :: ADialogue \rightarrow Score$ that returns the maximum total score you can obtain in a dialogue, where a score is the sum of the scores at each option chosen. Define $maxScore$ using the *fold* on the data type $ADialogue$ defined in Task 5. •

Solution 6.

```
maxScore :: ADialogue → Score
maxScore = foldDialogue maxScoreAlg where
  maxScoreAlg = (λc s d → d
                , λc (se1, a1, sc1) (se2, a2, sc2) → max (a1 + sc1) (a2 + sc2)
                , 0
                )
```

Marking

7 (10 points). Define a function $ppDialogue :: ADialogue \rightarrow (Int \rightarrow (String, Int))$ that prints an abstract dialogue in a way similar to how the example dialogues at the beginning of this exam are presented. The value of type $String$ in the resulting tuple is the printed dialogue. The Int that is passed in and returned in the computation is a number used to generate the labels for the parts of the paragraphs to which the commands in the choices jump. Define $ppDialogue$ using the *fold* on the data type $ADialogue$ defined in Task 5. •

Solution 7.

```
ppDialogue :: ADialogue → (Int → (String, Int))
ppDialogue = foldDialogue ppAlg where
  ppAlg = (λc s d → λi → let (s1, i1) = d i in (c ++ ": " ++ s ++ "\n" ++ s1, i1)
        , λc (se1, a1, sc1) (se2, a2, sc2) → λi →
          let (s1, i1) = a1 (i + 1)
              (s2, i2) = a2 (i1 + 1)
              c1      = if i1 == i + 1
                        then "[end] " ++ show sc1 ++ "\n"
                        else "[goto i" ++ show i ++ "]" ++ show sc1 ++ "\n"
              c2      = if i2 == i1 + 1
                        then "[end] " ++ show sc2 ++ "\n"
                        else "[goto i" ++ show i1 ++ "]" ++ show sc2 ++ "\n"
              l1      = if i1 == i + 1 then "" else "-i" ++ show i ++ "\n" ++ s1
              l2      = if i2 == i1 + 1 then "" else "-i" ++ show i1 ++ "\n" ++ s2
          in (c ++ ": " ++ "{" ++ se1 ++ "}" ++ c1 ++ "{" ++ se2 ++ "}" ++ c2 ++ l1 ++ l2
            , i2)
        , λi → (" [end] \n", i)
        )
```

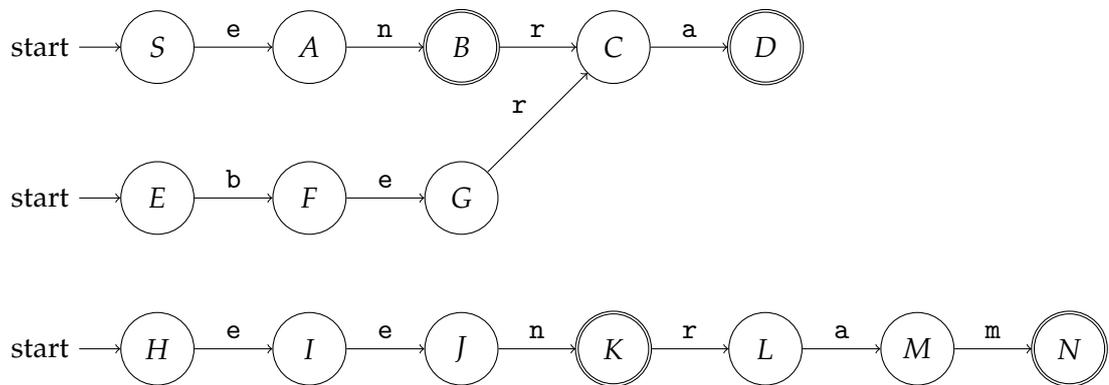
Marking

o

8 (5+5 points).

- (a) Give a nondeterministic finite automaton that recognizes the sentences "en", "een", "eenram", "enra", and "bera", and no more. The fewer states the better, but you don't have to construct an automaton with the absolute minimum number of states.

Solution 8.

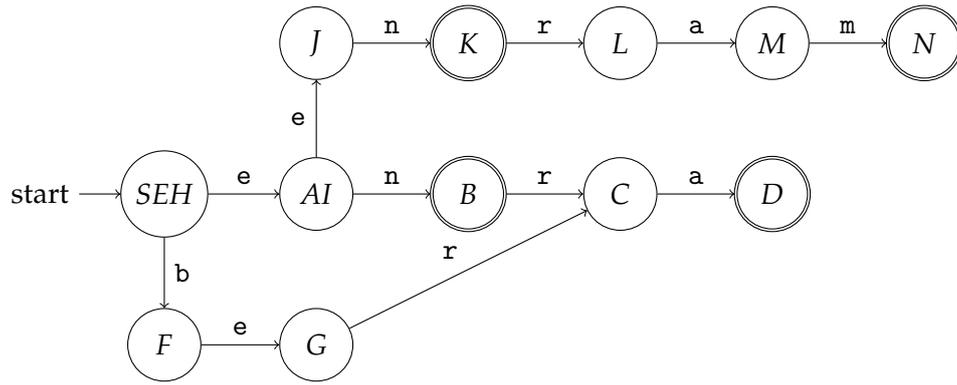


Marking

o

- (b) Is your nondeterministic automaton deterministic? If not, transform this nondeterministic automaton into a deterministic automaton using the standard transformation technique.

Solution 8.



Marking

-
-