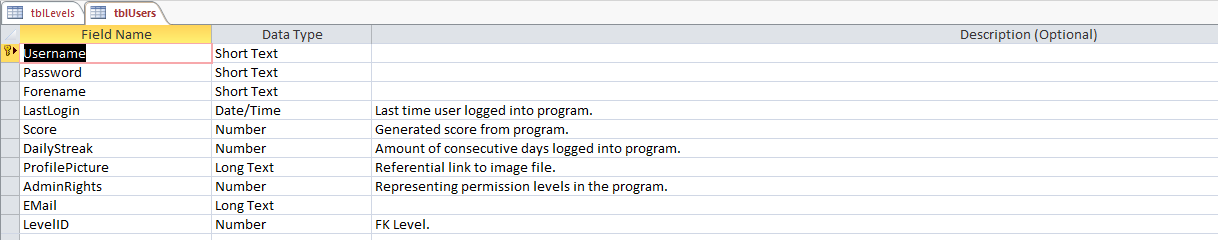
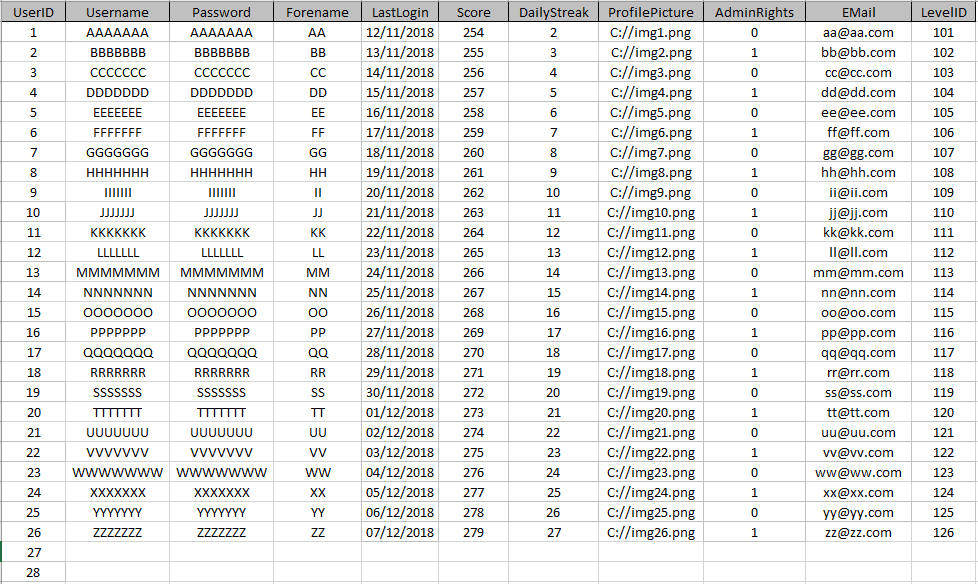
Development Diary

# 27/11/18

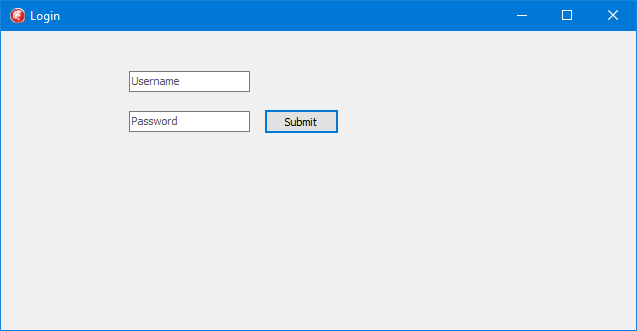
* Database created with relevant entities and fields. User table produced with the following fields:
* Validation rules added to each field where appropriate, e.g. for the username field:



Len([Username])>6

* Sample set of data created in excel, with each entity conforming to the validation rules:

This data will be used to test the functionality of the login screen. When the program is fully tested, the database will be populated with relevant entities for real users. This data will be sensitive and protected from unauthorized access.

* Basic login form created in Delphi in order to test the functionality of the user database table. I have only created the necessary objects for the form, so I will have to develop the code in the next session.

# 28/11/18

* Code behind the login form created.

procedure TloginForm.FormCreate(Sender: TObject);

var

DatabaseName:textfile;

text:string;

dlg:Topendialog;

filename:string;

connection:string;

connectstring:string;

begin

loginForm.Position:=poscreencenter;

assignfile(databasename,'filelocation.txt');

reset(databasename);

readln(databasename,text);

closefile(databasename);

showmessage(text);

try

if fileexists(text) then

connection:=text

else

begin

dlg:=topendialog.Create(nil);

try

if dlg.Execute() then

begin

filename:=dlg.filename;

showmessage(filename);

connection:=(filename);

rewrite(databasename);

writeln(databasename,connection);

closefile(databasename);

end

else

showmessage('No file has been selected.')

finally

dlg.Free();

end;

end;

finally

end;

connectstring:='provider=microsoft.ace.oledb.16.0;data source ='+connection+';persist security info=false;';

setconnectionstring(connectstring);

adotblUsers.Active:=false;

adotblusers.ConnectionString:=connectstring;

adotblusers.TableName:='tblUsers';

adotblusers.Active:=true;

end;

procedure TloginForm.FormShow(Sender: TObject);

begin

usernameEdit.Text:='';

passwordEdit.Text:='';

end;

procedure TloginForm.passwordEditKeyPress(Sender: TObject; var Key: Char);

begin

if ord(Key) = VK\_RETURN then

begin

Key := #0; // prevent beeping

submitButtonClick(Sender);

end;

end;

procedure TloginForm.submitButtonClick(Sender: TObject);

begin

if not ADOtblUsers.Locate('Username',usernameEdit.Text,[]) then

showmessage('Username or Password not in database. Seek administrator for further assistance.')

else

if passwordEdit.Text = ADOtblUsers['Password'] then

begin

setglobalpasslevel(ADOtblUsers['AdminRights']);

mainmenuForm.Show;

loginForm.Hide;

end

else

showmessage('Username or Password not in database. Seek administrator for further assistance.');

end;

This section of code is responsible for loading the database. When the user inputs their login credentials, they are checked against the user database and the main menu is shown if they are correct. Seek the testing document to view the functionality of this code.

* Code created to initialise global variables in Global Setups unit

unit Globalsetup;

interface

uses

sysutils;

var

globalconnectionstring:string;

globalpasslevel:integer;

procedure setconnectionstring(x:string);

procedure setglobalpasslevel(a:integer);

implementation

uses

Login;

procedure setconnectionstring(x:string);

begin

globalconnectionstring:=x;

end;

procedure setglobalpasslevel(a:integer);

begin

globalpasslevel:=a;

end;

end.

This section of code initialises and sets the connection-string and global pass level. The connection-string allows the program to create a link between the graphical forms and the database. This variable needs to be globalised since it is used within all of the forms. Similarly, the global pass level needs to be accessed within multiple forms so that elements appropriate to the user’s access rights are displayed.

* Basic code to initialize main menu form with tests for data-hiding

unit Mainmenu;

interface

uses

Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,

Vcl.Controls, Vcl.Forms, Vcl.Dialogs;

type

TmainmenuForm = class(TForm)

procedure FormCreate(Sender: TObject);

procedure FormShow(Sender: TObject);

procedure FormClose(Sender: TObject; var Action: TCloseAction);

private

{ Private declarations }

public

{ Public declarations }

end;

var

mainmenuForm: TmainmenuForm;

implementation

{$R \*.dfm}

uses

Login, Globalsetup;

procedure TmainmenuForm.FormClose(Sender: TObject; var Action: TCloseAction);

begin

loginForm.Show;

end;

procedure TmainmenuForm.FormCreate(Sender: TObject);

begin

mainmenuForm.Scaled:=true;

WindowState:=wsMaximized;

end;

procedure TmainmenuForm.FormShow(Sender: TObject);

begin

case globalpasslevel of

0:mainmenuForm.Caption:='Regular User Menu';

1:mainmenuForm.Caption:='Administrator Menu';

end;

end;

end.

# 30/11/18

* Code created for database encryption and password entry

procedure TloginForm.FormCreate(Sender: TObject);

var

DatabaseName: textfile;

text: string;

dlg: Topendialog;

filename: string;

connection: string;

connectstring: string;

password: string;

begin

InputQuery('Admin: Database Password',

'Enter the password for the database:', password);

connectstring := 'provider=microsoft.ace.oledb.16.0;data source =' +

connection + ';Jet OLEDB:Database Password=' + password + ';';

setconnectionstring(connectstring);

ADOtblUsers.Active := false;

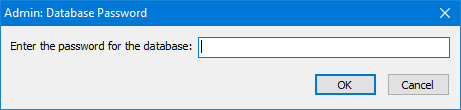
ADOtblUsers.ConnectionString := connectstring;

ADOtblUsers.TableName := 'tblUsers';

ADOtblUsers.Active := true;

end

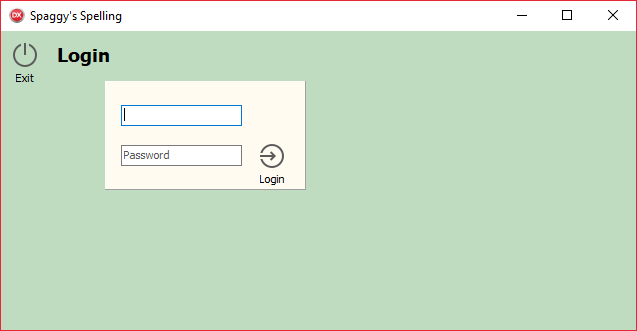
This prompts the admin user to enter a password that grants entry into the database. The database has been locked/encrypted with this password. Contents of the database cannot be gathered if this password is not correct.

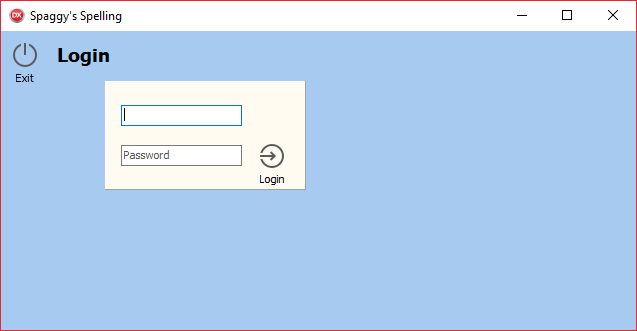


# 01/12/18

* All component names are altered to conform to Hungarian notation

For example, the name of the form ‘errorForm’ was changed to ‘frmError’. This is not a functional change to the program, but it does make the code more readable to other programmers, by following the universal conventions. The creation of new components such as forms will now be labelled appropriately.

* Simple UI features added to the login form

Firstly, in accordance to the success criteria (3: The solution will contain a graphical user interface which is aesthetically pleasing), colour was added to the form. As previously mentioned, bright contrasting colours are most favourable to a young target audience. After testing a series of different colour palettes, it was found that green and cream were the most aesthetic combo, according to a control group of two children, aged 5 and 6. Other colours were found to be too dark or ‘disgusting’. Blue and cream came in a close 2nd, which will most likely be used in other forms:

The second addition was the use of image icons in place of buttons. Another criticism made by the control group was that the buttons were too boring and stuck out amongst the bright colours. The compromise was to use icons which were sourced online[[1]](#footnote-1). This way the buttons will look more interesting and a continuity in design can be implemented, by sticking to the same icon package.

* Name added to program

In order to make the program more accessible to kids, the working title ‘Orthographic Assistance Program’ was changed to ‘Spaggy’s Spelling’. Spaggy will become the name of an easily identifiable mascot, a feature that was included in the Nessy program previously analysed. This name is purposefully silly and nonsensical to be amusing to young children.

# 03/12/18

* Implement daily streak and last login into login form

In order to keep tabs on how often the user signs in, the database must be updated on when the user logs in. This section of code was slotted into the submission click upon successful verification of the user:

adotblUsers.Locate('Username', edtUsername.text, []);

// Get daily streak from database and increment if last login was yesterday, otherwise reset

DailyStreak := adotblUsers.FieldByName('DailyStreak').AsInteger;

if adotblUsers.FieldByName('LastLogin').AsDateTime = Yesterday then

inc(DailyStreak)

else if adotblUsers.FieldByName('LastLogin').AsDateTime <> date then

DailyStreak := 0;

// Post amendments to the database and refresh

adotblUsers.Edit;

adotblUsers.FieldByName('DailyStreak').AsInteger := DailyStreak;

adotblUsers.FieldByName('LastLogin').AsDateTime := date;

adotblUsers.Post;

adotblUsers.Refresh;

frmMainmenu.Show;

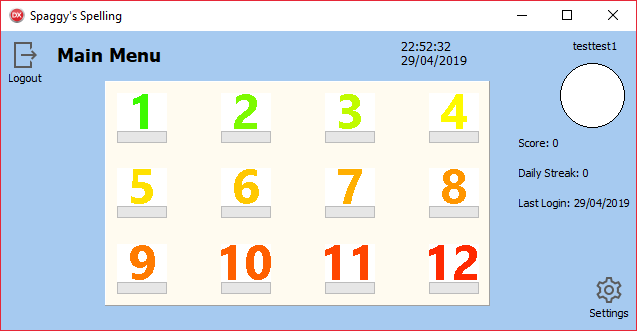
frmLogin.Hide;

In short, the program checks to see if the last login was yesterday and if so, increment the daily streak. If the last login was not yesterday, the daily streak is reset to 0. The streak is then updated in the database and the last login is set to the current date, therefore making it the ‘last login’.

This marks the first major iteration of development: the functional completion of the login form.

# 06/12/18

* Simple UI features and visual elements added to the main menu form



Following on from the control group testing, the blue and cream colour scheme was used, along with icons for the logout and settings buttons. This main menu layout is simple but easy to understand, with a main panel containing level buttons and a user side panel containing useful information. The code for data-hiding no longer just controls the caption of the form, but hides the admin options button as well:

// Display admin options to the user if they have administrator rights

case globalpasslevel of

0:

begin

frmMainmenu.Caption := 'Spaggy''s Spelling';

imgAdminOptions.Visible := false;

lblAdminOptions.Visible := false;

end;

1:

begin

frmMainmenu.Caption := 'Spaggy''s Spelling (Admin)';

imgAdminOptions.Visible := true;

lblAdminOptions.Visible := true;

end;

end;

# 10/12/18

* Code for displaying user information

with adoqryUsers do

begin

Active := false;

ConnectionString := globalconnectionstring;

SQL.Clear;

// Run SQL to gather all fields linked to the username

SQL.Add('Select \* from tblUsers where Username = "' +

globalusername + '";');

Active := true;

// Set focus to the record

First;

CurrentDir := GetCurrentDir;

// Set image directory (contained within program folder) from main program directory

// (SizeOf(Pointer)\*8) returns the windows platform at runtime (x86/x64)

ImageDir := StringReplace(CurrentDir, 'Win' + (inttostr(SizeOf(Pointer) \* 8)

) + '\Debug', 'Profile Pictures\', [rfIgnoreCase]);

// Load the user's profile picture to imgProfilePic

imgProfilePic.Picture.LoadFromFile

(ImageDir + adoqryUsers['ProfilePicture']);

// Load the user's profile info and display within corresponding labels

lblScore.Caption := 'Score: ' + floattostr(adoqryUsers['Score']);

lblDailyStreak.Caption := 'Daily Streak: ' +

floattostr(adoqryUsers['DailyStreak']);

lblLastLogin.Caption := 'Last Login: ' +

datetostr(adoqryUsers['LastLogin']);

// Store user's level progress to local array

for i := 1 to 12 do

UserProgress[i] := FieldByName('Level' + inttostr(i)).AsInteger;

end;

lblUsername.Caption := globalusername;

This section of code uses the adoquery element to gather all fields associated with the user using the SQL query 'Select \* from tblUsers where Username = user;’ The profile pic is loaded first, requiring the generation of the image directory. Since clients have the option of using the 32bit or 64bit version, the image directory is dependent on which version is currently in use. Next, the score, daily streak and last login are loaded into the appropriate label captions. Finally, the user progress from the database is stored into a local array, which will later be implemented into the progress bars on the form.

# 14/12/18

* Progress bar implementation

For the user to track their progress, a progress bar was put at the bottom of each level. This shows how far the user is into each level at a glance.

for x := 1 to 2 do

// Set positions for each progress bar in the main menu according to level progress

for i := 1 to 12 do

begin

pb := TProgressBar(frmMainmenu.FindComponent('pb' + inttostr(i)));

pb.Position := trunc((UserProgress[i] / LexiconSize[i]) \* 100);

// Set progress bar colour to green if full (completed level)

if pb.Position = 100 then

pb.State := pbsNormal;

end;

This code sets the position of each progress bar to the percentage of the user progress over the size of the level. If the position is at 100, the colour of the progress bar is set to green, to clearly show the level is finished. Initially, this code was much larger, as each of the 12 progress bars had to be handled individually. The inclusion of a for loop makes this code more concise, although it required extra research to find each component. Furthermore, due to the limitations of object programming in Delphi, the setting of progress bar positions had to be completed twice. Completing it once does not always work reliably.

procedure TfrmMainmenu.imgLvl2Click(Sender: TObject);

begin

if ((pb1.Position > 10) and (pb2.Position < 100)) then

begin

setglobalgamelevel(2);

frmLevel.Show;

frmMainmenu.Hide;

end

else

showmessage('Level is locked. Try playing a previous level.');

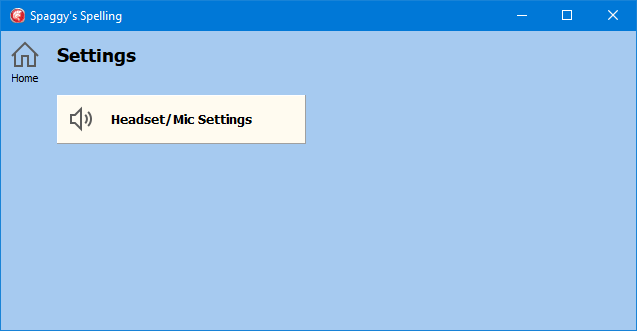
end;

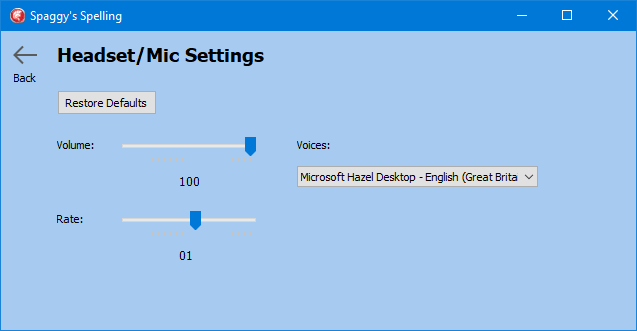
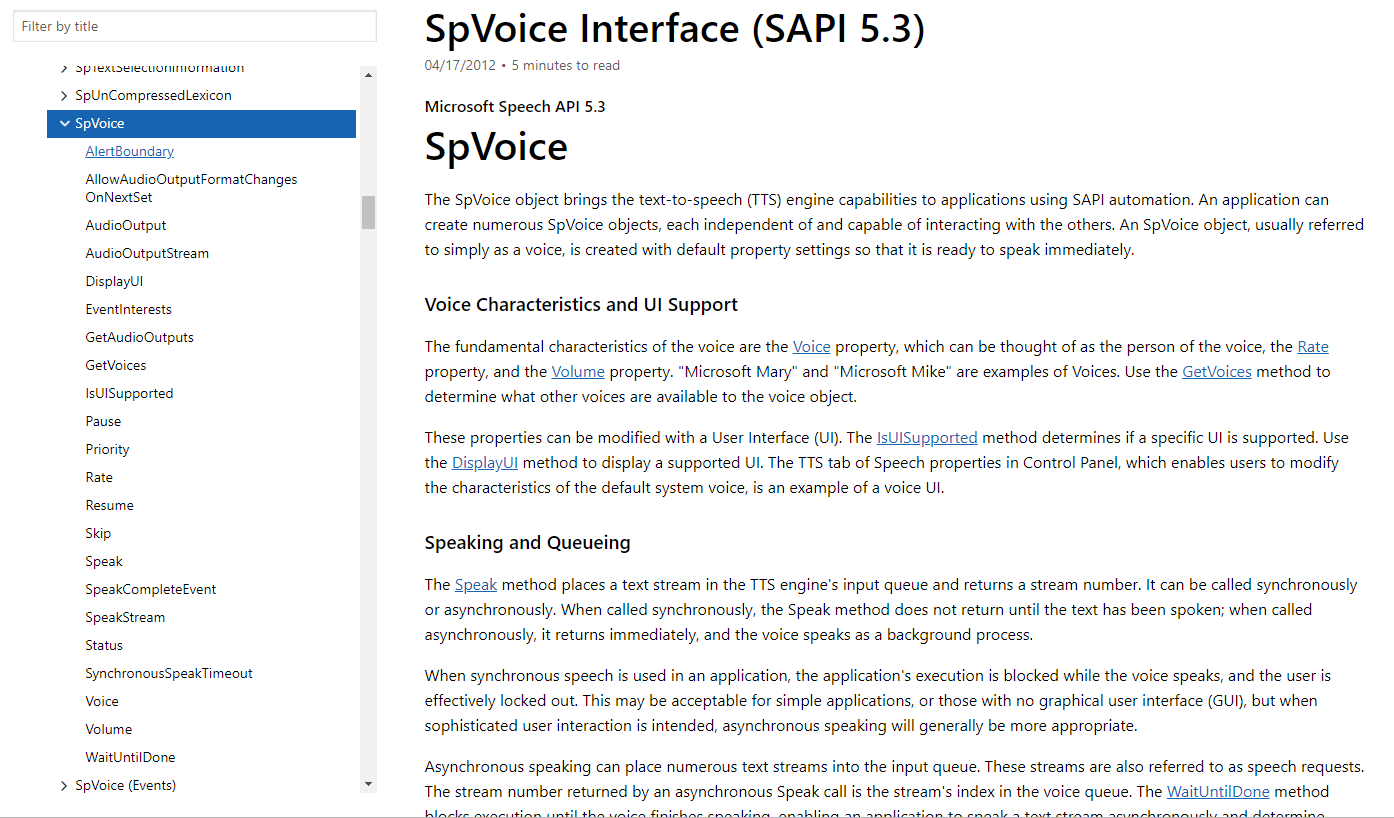
Upon selecting a level, an initial check is made to see if the level is incomplete and that a previous level has been partially completed. This decision was made to aid in the natural progression of the game, preventing users from immediately jumping into the last level.

Although selecting a level does not yet do anything, the main menu is functional. Therefore, the next major iteration has been met.

# 15/12/18

* Create forms and UI elements for the headset/mic settings

The only customizable elements for my program are those attached to the speech API, including speech rate and volume. Later development could include settings for other elements of the program. For example, display settings to change the colour scheme, making the program easier to see for those with colour blindness. For now, forms will be created for the base settings menu and the headset/mic settings.

With reference to the official Microsoft SAPI ver5.3 documentation[[2]](#footnote-2), a decision was made to add customization to the rate, volume and voice properties. For the voice property, a combo box was created to provide the user with all the voices available to them on their system. This section of code populates this combo box:

procedure TfrmHeadsetsettings.FormCreate(Sender: TObject);

var

i: integer;

begin

frmHeadsetsettings.Position := poscreencenter;

// Gather speech object tokens for all available voices (SpeechAPI)

SOTokens := SpVoice.GetVoices('', '');

// Populate cbVoices with voices

for i := 0 to (SOTokens.Count - 1) do

begin

SOToken := SOTokens.Item(i);

cbVoices.Items.AddObject(SOToken.GetDescription(0), TObject(SOToken));

//Increment descriptor reference count to ensure it's not destroyed

SOToken.\_AddRef;

end;

end;

The SAPI documentation also refers to how speech voices are stored and identified, using ‘Speech Object Tokens’ or ‘SOTokens’. The line SpVoice.GetVoices(‘’,’’) is responsible for gathering all available speech voices and storing them in an array. The for loop then goes through this array and adds each voice to the combo box, making sure to add them as an object and not a string. The object description refers to the string of text that appears to the user (‘Microsoft Hazel Desktop’).

Considerable time was spent coming up with a way to transfer the user’s configuration to the level form, where the speech object actually resides. Making changes to the speech object in the HeadsetSettings form does not make the same changes to the speech object in the level form, as they are considered separate objects by the program. The best solution was to store the user’s configuration in a config file, which could be accessed from the level form.

procedure TfrmHeadsetsettings.FormShow(Sender: TObject);

begin

CurrentDir := GetCurrentDir;

LexiconDir := StringReplace(CurrentDir, 'Win' + (inttostr(SizeOf(Pointer) \* 8)

) + '\Debug', 'Lexicon\', [rfIgnoreCase]);

// Create TStringList object, and populate the list with the values in the

// Config file: 1st line = volume .. etc...

sl := TStringList.Create;

sl.LoadFromFile(LexiconDir + 'VoiceConfig.txt');

tbVolume.Position := strtoint(sl[0]);

tbRate.Position := strtoint(sl[1]);

cbVoices.ItemIndex := strtoint(sl[2]);

end;

This code is performed when the form is loaded up by the user. It populates a string list of the current settings from the config file and loads them onto the form through the track bars and the combo box.

# 19/12/18

* Code created to link the setting elements to the config file

procedure TfrmHeadsetsettings.tbRateChange(Sender: TObject);

begin

// On change of rate, post new rate to config file

lblTBRatePos.Caption := (formatfloat('00', tbRate.Position));

sl[1] := inttostr(tbRate.Position);

sl.SaveToFile(LexiconDir + 'VoiceConfig.txt');

end;

Through the event ‘onChange’, each setting element is linked to the config file. When the user changes the setting, the string list is updated and saved to the config file.

procedure TfrmHeadsetsettings.btnRestoreClick(Sender: TObject);

begin

// Set trackbars and combobox to default values

// This will trigger onChange event for each, posting changes to config file

tbVolume.Position := 100;

tbRate.Position := 0;

cbVoices.ItemIndex := 0;

sl[2] := inttostr(cbVoices.ItemIndex);

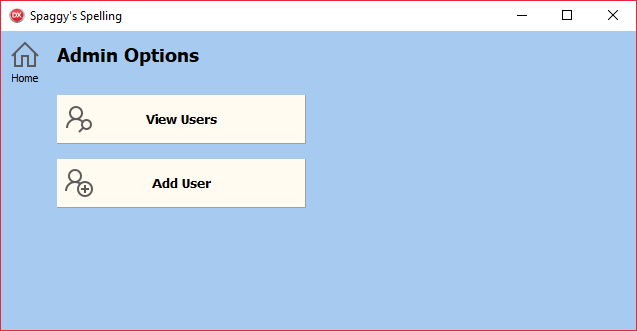
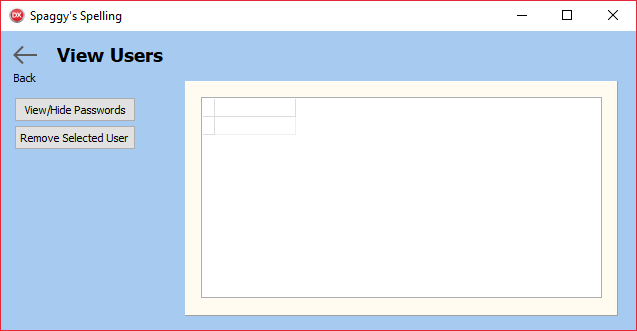
sl.SaveToFile(LexiconDir + 'VoiceConfig.txt');

end;

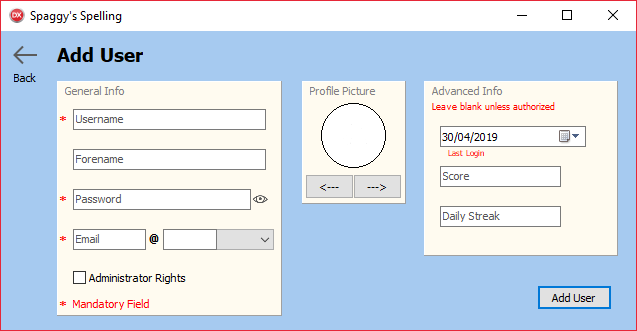
When the restore defaults button is pressed, the elements are all reset to their default values and the config file is updated to reflect this. The form is now completed.

# 21/12/18

* Create forms and UI elements for the admin options

For this program to follow the success criteria, it has to follow the rules of CRUD, in creating, reading, updating and destroying data. Data will be updated in the level itself, through tracking the user’s progress. These admin options complete the other 3 rules. The first is reading data:

This form is setup using the dbGrid, to show all fields in the user table of the database. This grid will be configured in later development. Buttons have also been added to view/hide passwords, which will require an extra step of verification, and to remove users, which gives the option to destroy data.

This form contains all the input boxes necessary to add another user/field to the user table, therefore giving the option to create data. Mandatory fields are those fields with validation rules attached to them. Verification of these fields will have to be within the form so that the database doesn’t return any errors, which will appear when data entry doesn’t conform to the validation rules.

# 06/01/19

* Link the dbGrid in the ViewUsers form to the database and create the code for the buttons to function properly

procedure TfrmViewusers.FormShow(Sender: TObject);

var

i: integer;

ColumnArray: array of integer;

begin

adotblUsers.ConnectionString := globalconnectionstring;

adoqryDelete.ConnectionString := globalconnectionstring;

adotblUsers.TableName := 'tblUsers';

adotblUsers.Active := true;

// Only show neccessary columns in the database, e.g. hide user passwords

ColumnArray := [1, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20];

for i := 0 to 9 do

dbGrid.Columns[i].Width := 80;

for i in ColumnArray do

dbGrid.Columns[i].Visible := false;

end;

This section of code sets the connection strings of the ado objects in the form, which the dbGrid is connected to. During testing, it was found that the dbGrid was too small to correctly show all the columns and fields. The first step in mitigating this issue was to only show the necessary columns that the admin would need access to. This meant skipping out on score, last login, level progress and passwords. Since this grid is only supposed to give an overview of users, it makes sense to only show columns directly relating to the users. If an admin has to edit fields not shown on this grid then they can still do so from within the database itself. Passwords are not initially shown to apply an extra layer of verification:

procedure TfrmViewusers.sbtnViewPasswordClick(Sender: TObject);

var

password: string;

begin

// Only show user passwords once admin has correctly entered db password

InputQuery('Admin: Database Password', 'Enter the password for the database:',

password);

if password = globaldbpassword then

dbGrid.Columns[1].Visible := sbtnViewPassword.Down

else

showmessage('Database Error: Incorrect database password.');

end;

A speed button was used for this operation to act as a toggle switch: viewing and hiding. This code presents an input box to the admin, where they enter the password for the database. Globaldbpassword is a variable within the Globalsetup form that is set when the admin first correctly enters the database password. If the password is correct, the password column in the grid is set to the state of the button. Upon pressing the button for the first time, visible is set to true.

procedure TfrmViewusers.btnRemoveUserClick(Sender: TObject);

var

password: string;

buttonSelected: integer;

user: string;

begin

// Only remove user if admin has correctly entered db password

InputQuery('Admin: Database Password', 'Enter the password for the database:',

password);

if password = globaldbpassword then

begin

// Set user variable to the username of the selected row on dbGrid

user := dbGrid.Fields[0].AsString;

// Present user with confirmation dialog

buttonSelected := messagedlg('Confirm that the user "' + user +

'" will be REMOVED from the database.', mtCustom, mbOKCancel, 0);

// Locate and delete user if user presses ok on confirmation dialog

if buttonSelected = mrOK then

begin

dbGrid.DataSource.DataSet.Locate('Username', user, []);

dbGrid.DataSource.DataSet.Delete;

showmessage('"' + user + '" has been removed from the database.');

end

else

showmessage('"' + user + '" has NOT been removed from the database.');

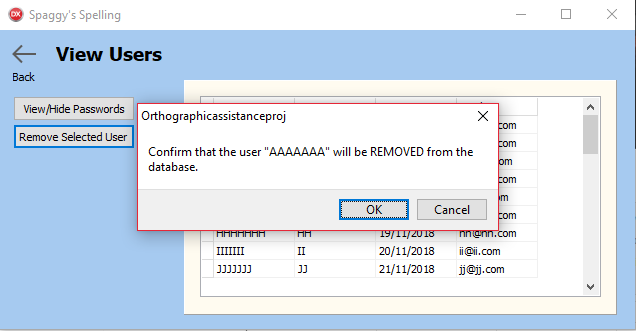
end

else

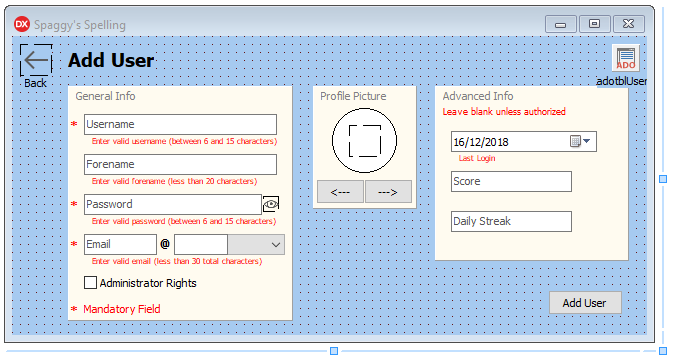
showmessage('Database Error: Incorrect database password.');

end;

Upon selecting the ‘remove user’ button, the same verification as the ‘view passwords’ button is performed. When verified, the currently selected field is set to the user that will be deleted. Since this is a sensitive action that results in loss of data, another check is performed to confirm that the admin wants to delete that user. A message dialog box is presented:

Upon confirmation, the user is located through the adotable and deleted. If cancel is pressed, no further action is taken.

# 09/01/19

* Add verification to data entry in the AddUser form and create code to add user to the database

This screenshot shows the design view within Delphi. The red labels under the edit boxes were added to represent the corresponding verification rules in the database. These are shown upon invalid data entry.

At this point, profile pictures were incorporated into the program. It was decided that letting the user upload their own images would bring in too much discrepancy. As a compromise, in order to maintain the customization of user profiles, a selection of 14 different pictures were designed in photoshop:



(+male versions)

Although these pictures are simple, they are reflective of all diversities and will give an illusion of customization for the young target audience. All 14 pictures are stored locally to the program (essentially pre-installed), so no further alteration is required by the client of the program. The buttons in the form could then be set up to cycle through these images.

procedure TfrmAdduser.btnProfilePicRightClick(Sender: TObject);

begin

if CurrentPic = 14 then

CurrentPic := 1

else

Inc(CurrentPic);

imgProfilePic.Picture.LoadFromFile(ImageDir + inttostr(CurrentPic) + '.png');

end;

procedure TfrmAdduser.btnProfilePicLeftClick(Sender: TObject);

begin

if CurrentPic = 1 then

CurrentPic := 14

else

Dec(CurrentPic);

imgProfilePic.Picture.LoadFromFile(ImageDir + inttostr(CurrentPic) + '.png');

end;

CurrentPic is a variable that keeps track of the picture currently being displayed. The pictures also contain filenames consisting only of numbers between 1 and 14, allowing the program to cycle through them.

# 14/01/19

* Continuation of validation to data entry

procedure TfrmAdduser.btnAddUserClick(Sender: TObject);

var

i: Integer;

Error: boolean;

vInt: Integer;

vStr: string;

begin

// Set Error variable to false. If there is one error in input validation then Error is true

// Input validation corresponds to the validation rules set in the database (or if input blank)

// For each input error, display error hint to user and set focus to the input box

Error := false;

if (edtUsername.Text = '') or (Length(edtUsername.Text) < 6) or

(Length(edtUsername.Text) > 15) then

begin

lblUsernameError.Visible := true;

edtUsername.SetFocus;

Error := true;

end

else

lblUsernameError.Visible := false;

if (Length(edtForename.Text) > 20) then

begin

lblForenameError.Visible := true;

edtForename.SetFocus;

Error := true;

end

else

lblForenameError.Visible := false;

if (edtPassword.Text = '') or (Length(edtPassword.Text) < 6) or

(Length(edtPassword.Text) > 15) then

begin

lblPasswordError.Visible := true;

edtPassword.SetFocus;

Error := true;

end

else

lblPasswordError.Visible := false;

if (edtEmailPre.Text = '') or (edtEmailSuf.Text = '') or

(cbEmail.ItemIndex = -1) or

((Length(edtEmailPre.Text) + Length(edtEmailSuf.Text) + 6) > 30) then

begin

lblEmailError.Visible := true;

edtEmailPre.SetFocus;

Error := true;

end

else

lblEmailError.Visible := false;

if not Error then

(..)

This long section of code is responsible for validating all the inputs. These ‘if’ statements work around the central ‘Error’ boolean variable. If the input validation is incorrect for at least one input then error will return true. All inputs must be inputted correctly to proceed. When an error is encountered, the corresponding error label is shown to the admin to inform them of their mistake and the mouse cursor is set to that position.

if not Error then

begin

adotblUsers.First;

// Check new username against existing usernames. If it already exists, raise an error

repeat

if uppercase(adotblUsers['Username']) = uppercase(edtUsername.Text) then

Error := true;

adotblUsers.Next;

until adotblUsers.Eof or Error;

// If there are no validation errors and username is unique then append new record

if not Error then

begin

adotblUsers.Close;

adotblUsers.Open;

adotblUsers.Append;

adotblUsers['Username'] := edtUsername.Text;

adotblUsers['Password'] := edtPassword.Text;

adotblUsers['Forename'] := edtForename.Text;

adotblUsers['LastLogin'] := dtpLastLogin.Date;

// Only append advanced info if entered (optional field)

if edtScore.GetTextLen > 0 then

adotblUsers['Score'] := strtoint(edtScore.Text)

else

adotblUsers['Score'] := 0;

if edtDailyStreak.GetTextLen > 0 then

adotblUsers['DailyStreak'] := strtoint(edtDailyStreak.Text)

else

adotblUsers['DailyStreak'] := 0;

adotblUsers['ProfilePicture'] := inttostr(CurrentPic) + '.png';

if ckbxAdmin.Checked then

adotblUsers['AdminRights'] := 1

else

adotblUsers['AdminRights'] := 0;

adotblUsers['EMail'] := edtEmailPre.Text + '@' + edtEmailSuf.Text +

cbEmail.Text;

(..)

This code first checks to see that the chosen username is unique. Since the username is a primary key in the user table, this value must not have any duplicates. If there are no validation errors and the username is unique, a new record is appended to the database with the admin’s inputs. Optional fields are only appended if they have been changed from their defaults by the admin. The Email field pulls together inputs from the prefix, suffix and domain check box.

(..)

// Repeatedly present input box until a valid school year is entered

// This completes levels that are below the user's current level of knowledge

repeat

repeat

vStr := InputBox('User Info',

'Enter the school year the user is currently attending (0 for reception, max year 5):',

'0');

until TryStrToInt(vStr, vInt);

until (vInt >= 0) and (vInt <= 5);

// Set user progress to default (0 for each level)

for i := 1 to 12 do

adotblUsers['Level' + inttostr(i)] := 0;

// Depending on user input, mark levels below their current knowledge as complete

for i := 1 to (vInt \* 2) do

adotblUsers['Level' + inttostr(i)] := LexiconSize[i];

// Post changes to database and reload

adotblUsers.Post;

adotblUsers.Refresh;

// Inform admin that the user has been successfully added (not an error)

setglobalerrorcode('addusernotification.txt');

frmError.Show;

ClearForms;

end

else

begin

setglobalerrorcode('addusernametakenerror.txt');

frmError.Show;

end;

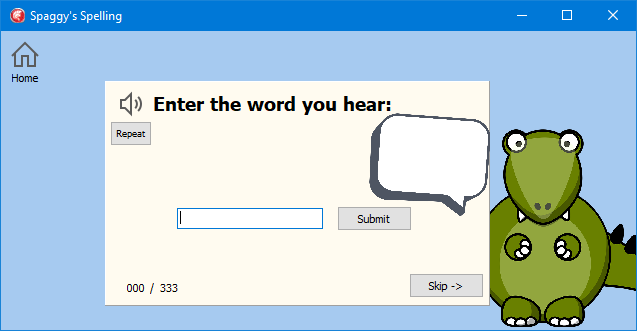
In accordance to the success criteria (2: The solution will provide a tailored learning experience, which will be unique to each individual user), extra code was added to connect the user’s previous knowledge of spellings to the equivalent progress within the game. An input box is presented to the admin to enter the school year that the user is currently attending. Since there are 12 levels for 6 years of school (including reception), multiplying the input by 2 and looping through all levels up until this input sets the user’s current progress. For example, if the student currently attended year 2, the for loop would mark levels 1,2,3 and 4 as already completed. This tailors the learning experience for each user.

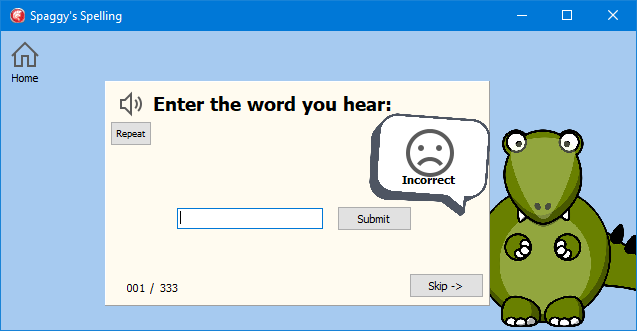
After this, the changes made to the database are updated and a notification is shown to inform the admin. This marks the third major iteration of development in completing the admin options. The only part of CRUD that is not met by the program is updating data, and this will be performed in the level form.

# 20/01/19

* Create form and UI elements for the game level

This form design is heavily based upon the prototype made as part of the design. It is clearly laid out; with the main central panel, containing all functions for the game level, and a home button. Its simplicity makes the program easy to follow, especially for the young target audience. The mascot has also been added to the side of the screen, to give the users a recognisable figure to follow.

When a mistake is made in the level, an unhappy face is shown, along with ‘Incorrect’:



* Create the code necessary to load the lexicon game file and the SAPI config file

procedure TfrmLevel.FormShow(Sender: TObject);

var

i: integer;

begin

// Using ADO object, locate the user's data and load previous level progress to count

with adotblUsers do

begin

Active := false;

ConnectionString := globalconnectionstring;

TableName := 'tblUsers';

Active := true;

Open;

Locate('Username', globalusername, []);

count := FieldByName('Level' + inttostr(globalgamelevel)).AsInteger;

end;

// Reset GUI labels

guesses := 0;

lblResult.Caption := '';

imgResult.Visible := false;

lblProg1.Caption := formatfloat('000', count);

lblProg3.Caption := inttostr(LexiconSize[globalgamelevel]);

CurrentDir := GetCurrentDir;

// Set icon and lexicon directory (contained within program folder) from main program directory

// (SizeOf(Pointer)\*8) returns the windows platform at runtime (x86/x64)

IconDir := StringReplace(CurrentDir, 'Win' + (inttostr(SizeOf(Pointer) \* 8)) +

'\Debug', 'Icons\', [rfIgnoreCase]);

LexiconDir := StringReplace(CurrentDir, 'Win' + (inttostr(SizeOf(Pointer) \* 8)

) + '\Debug', 'Lexicon\', [rfIgnoreCase]);

// Create TStringList object, and populate the list with the values in the

// Config file: 1st line = volume .. etc...

sl := TStringList.Create;

sl.LoadFromFile(LexiconDir + 'VoiceConfig.txt');

// Set volume and rate of SpVoice to values in config file (SpeechAPI)

SpVoice.Volume := strtoint(sl[0]);

SpVoice.Rate := strtoint(sl[1]);

cbVoices.ItemIndex := strtoint(sl[2]);

// Set voice of SpVoice to the object in cbVoices that corresponds to the

// Index in the config file

SOToken := ISpeechObjectToken

(Pointer(cbVoices.Items.Objects[cbVoices.ItemIndex]));

SpVoice.Voice := SOToken;

(..)

When the level is selected from the main menu, the variable globalgamelevel is defined in the Globalsetup unit, holding the level currently being played. This section of code takes that game level and sources the user’s current progress within it. For example, the user may have partially completed a level and come back to it. It then defines the two numbers at the bottom left of the game panel, showing the progress into the level. After this, the code loads the SAPI config file and applies these configurations to the SpVoice object. This object is responsible for the text-to-speech feature.

(..)

// Assign the current game level file from the lexicon directory to the variable Lexicon

assignfile(Lexicon, LexiconDir + inttostr(globalgamelevel) + '.txt');

reset(Lexicon);

// Set the current word by moving through the words before it in the lexicon

// This will depend on current level progress through 'count'

for i := 0 to count do

readln(Lexicon, word);

// Using the SpVoice component of the SpeechAPI package, output the word to be tested

// Use asynchronous flag to return the command as soon as the speak request is queued

SpVoice.Speak(word, 1);

end;

This section of code moves through the flat file lexicon to the currently tested word and the SpVoice object outputs the word through the user’s audio device.

# 23/01/19

* Apply functions to the submit, skip and repeat buttons

procedure TfrmLevel.btnSubmitClick(Sender: TObject);

begin

guess := edtGuess.Text;

edtGuess.Text := '';

lblAnswer.Visible := false;

imgResult.Visible := true;

// If the guess matches the word then update progress and move onto next word

if UpperCase(guess) = UpperCase(word) then

begin

// Display image informing the user that their guess is correct

imgResult.Picture.LoadFromFile(IconDir + 'icons8\_Happy\_64px.png');

lblResult.Caption := 'Correct';

inc(count);

UpdateProgress(count, true);

// If 'count' is equal to the size of the current lexicon then the level is complete

if count = LexiconSize[globalgamelevel] then

begin

showmessage('Level Completed.');

frmLevel.Close;

end

else

// Else speak the next word

begin

guesses := 0;

readln(Lexicon, word);

SpVoice.Speak(word, 1);

edtGuess.SetFocus;

end;

end

else

// Else display image informing the user that their guess is incorrect

begin

imgResult.Picture.LoadFromFile(IconDir + 'icons8\_Sad\_64px.png');

lblResult.Caption := 'Incorrect';

// Increment guesses for each incorrect guess

inc(guesses);

// If the user makes 3 failed attempts then display correct answer

if guesses = 3 then

begin

lblAnswer.Visible := true;

edtGuess.Text := word;

imgResult.Visible := false;

lblResult.Caption := '';

end;

SpVoice.Speak(word, 1);

edtGuess.SetFocus;

end;

end;

This section of code takes the input of the edit box and compares it to the word in the lexicon. If both words are the same (case insensitive), then the user has correctly spelled the word and they are presented with a happy face. In order to reduce the size of this procedure, the updating of progress is handled in a separate procedure. This is called within the skip button as well, so duplicate code has been avoided. From here, if the value of count is equal to the lexicon size, a constant global array holding the amount of words in each lexicon, then the level is complete and the form is automatically closed. Otherwise, the next word is spoken. If the guess made by the user is incorrect, the holding variable ‘guesses’ is incremented. This gives the user 3 guesses before being presented with the correct answer. In practice, the user would come back to repeat this level knowing the word they didn’t know previously. This marks a progression of knowledge, which this program is trying to achieve.

procedure TfrmLevel.btnRepeatClick(Sender: TObject);

begin

SpVoice.Speak(word, 1);

edtGuess.SetFocus;

end;

procedure TfrmLevel.btnSkipClick(Sender: TObject);

begin

// Increment count and update user's data, skipping the current word

inc(count);

UpdateProgress(count, false);

// Speak next word

readln(Lexicon, word);

SpVoice.Speak(word, 1);

lblResult.Caption := '';

imgResult.Visible := false;

edtGuess.SetFocus;

end;

The skip button simply increments the variable count, which holds the position in the game, and updates the progress with the parameter false, informing the UpdateProgress procedure to not increment the user’s ‘score’ value in the database. Therefore, users who have more correct guesses and less skips will have a higher score than those who don’t. Progress is not updated here as it is deemed that the user does not know how to spell the word.

The repeat click speaks the word again and sets the focus to the edit box. This will be used when the user didn’t catch the word being said, and using this button has no bearing on the progress into the game.

# 24/01/19

* Complete the UpdateProgress procedure to upload the user’s progress to the database

// UpdateProgress procedure updates the user's data for current level progress

// 'a' parameter is false for skipped answers, to correctly reflect user's score

procedure TfrmLevel.UpdateProgress(x: integer; a: boolean);

var

score: integer;

begin

adotblUsers.Edit;

// Set the value of the current level progress to x

adotblUsers.FieldByName('Level' + inttostr(globalgamelevel)).AsInteger := x;

score := adotblUsers.FieldByName('Score').AsInteger;

// Only increment user's score if answer hasn't been skipped

if a then

adotblUsers.FieldByName('Score').AsInteger := (score + 1);

adotblUsers.Post;

adotblUsers.Refresh;

// Display current level progress to the user

lblProg1.Caption := formatfloat('000', x);

end;

This procedure sets the current level progress in the database to the first parameter, being the updated level progress in the game. It then only increments the user’s score if the second parameter is set to true, which is called when the word is correctly guessed. Finally, the level progress is presented to the user on the bottom left of the main panel.

This marks the final iteration in development, with a complete, working program that will now be tested and evaluated. The solution conforms to CRUD and meets several success criteria.

1. icons8.com – windows 10 Icons [↑](#footnote-ref-1)
2. https://docs.microsoft.com/en-us/previous-versions/windows/desktop/ms723602(v%3dvs.85) [↑](#footnote-ref-2)