The Busby App

In this lab, you will work on developing the Busby app. This is a chatbot for finding information about realtime bus arrivals. The goal is to help you understand how to use the SG APIs in a simple app.

The Busby app is entirely contained in the `busby-vX.m` files. The app comes in 3 versions, of increasing complexity:

1. `busby-v0.m` - lists bus arrivals in minutes for a given bus stop code. You will learn how to get this information (using `smojo/api/sg/gov/bus`) and how to display it.
2. `busby-v1.m` - has the same function as v0, but additionally gives the user an option of locating nearby bus stops. This uses the `smojo/api/sg/gov/bus/info` cloud word.
3. `busby-v2.m` - the same function as v1, but additionally plots the nearest bus stops on a map. You will use the `smojo/api/sg/gov/map` word for this.

I've assumed that you already know a bit about Smojo, but I will explain more here.

API Documentation

All APIs are documented using the standard `smojo/doc` cloud word. The 4 API covered are:

1. **`smojo/api/sg/gov/bus`** - Realtime bus information. This is hosted by Singapore's Land Transport Authority (LTA).
2. **`smojo/api/sg/gov/bus/info`** - Information on bus stops and routes. This information is reformatted from the LTA database to make it more convenient for chatbot creators to use. It is hosted by Terra AI.
3. **`smojo/api/sg/gov/map`** - Mapping API and address search. Hosted by OneMap.
4. **`smojo/api/sg/gov/places`** - Information on various places of interests in Singapore, including F&B etc. Hosted by the Singapore Tourism Board.

To obtain documentation, use the **`doc`** word after importing `smojo/doc` . For example, if you want the documentation on **`smojo/api/sg/gov/places`**

```
smojo/doc

doc smojo/api/sg/gov/places
```

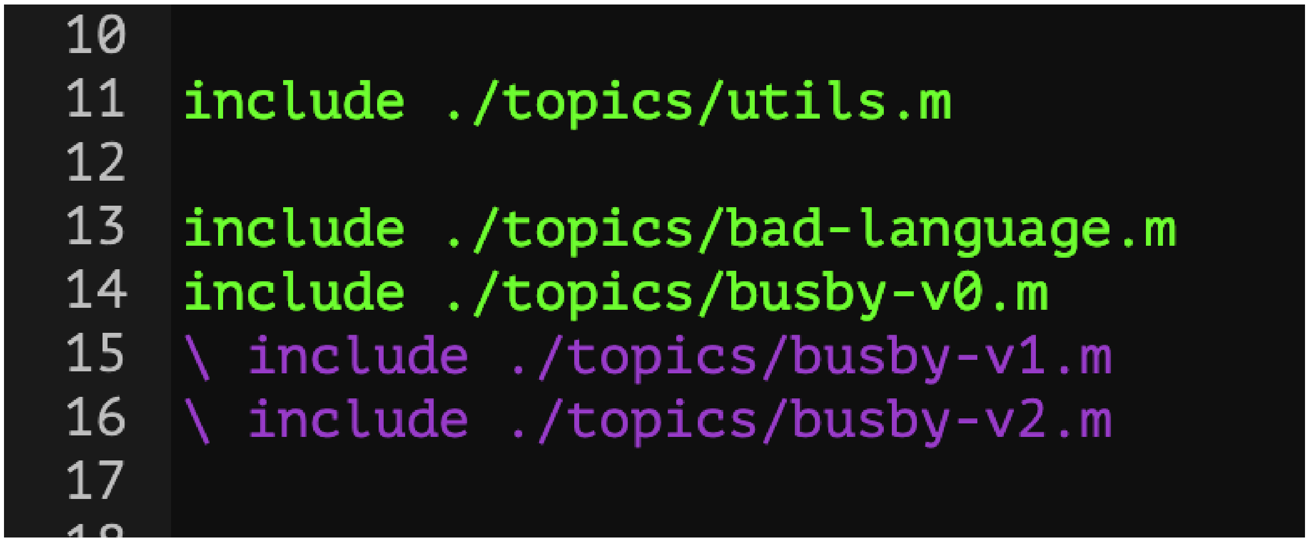
And click on "Run" (the green play button). The documentation should appear on the output panel. You should try this now and become very familiar with finding out information using `doc` .

Pro TIP: The docs also have sample code that you can cut-and-paste into the IDE and hit "Run" to execute.

Busby V0

Let's get Busby v0 working!

Step 1: Open `main.m` and ensure that just busby-v0.m is included (see Fig 1 below).



```

include ./topics/utils.m

include ./topics/bad-language.m
include ./topics/busby-v0.m
\ include ./topics/busby-v1.m
\ include ./topics/busby-v2.m

```

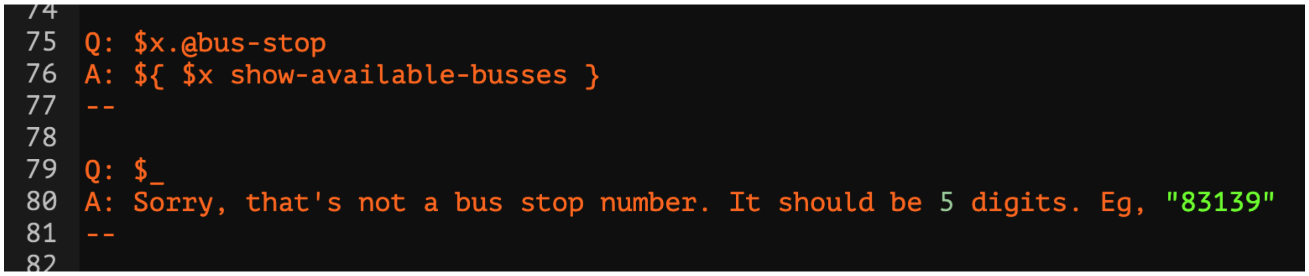
Figure 1: Be careful to check that only one version is included at any time!

Step 2: `publish` the app and test as usual.

Understanding Busby v0

Once you're done playing with this app, open up `busby-v0.m` and let's examine the contents:

Scroll right down, there are just 2 templates:



```
Q: $x.@bus-stop
A: ${ $x show-available-busses }
--

Q: $_
A: Sorry, that's not a bus stop number. It should be 5 digits. Eg, "83139"
--
```

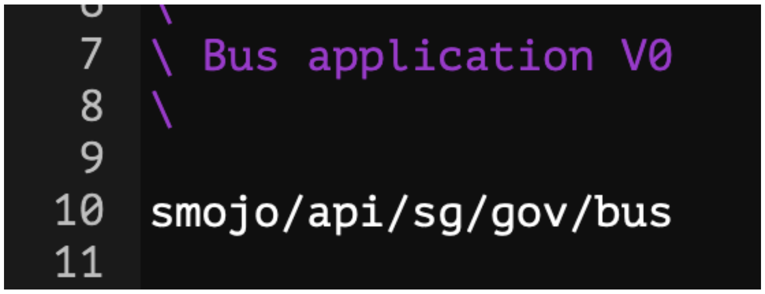
Figure 2: The two templates of v0

1. The first template checks for a bus-stop code (a 5-digit number). This is done using a **programmatic type**, called `@bus-stop` farther up the file. The most important word is `show-available-busses ( "bus-stop-code" -- "s" )`, which I will dig into shortly.
2. The second template is a catchall that gives the user a suggestion for a possible bus stop code.

show-available-busses

The `show-available-busses` word takes the bus-stop code as input and returns a HTML table showing all arriving busses at that stop.

To do this, the word uses the API word `bus-arrival-all` from the `smojo/api/sg/gov/bus` . This is why the top of `busby-v0.m` needs an import of this cloud word:



```
\ Bus application V0
\

smojo/api/sg/gov/bus

```

Figure 3: Importing the API cloud word import

Exercise 1: Use the `doc` word to find out more about `bus-arrival-all` . What are the inputs? What are the outputs?

`bus-arrival-all` outputs a hash.

1. The **keys** of the hash are the `bus-numbers` ,
2. and the **values** are a **sequence** of **`bus`** objects. The API documentation tells you what properties are available on these **`bus`** objects. The most important one for us is `estimated-arrival-mins@` , which gives the estimated arrival time in minutes.

Note that a sequence of bus objects is returned, ordered by time of arrival. The first item on the list is the earliest bus to arrive.

So, to display all the busses and their earliest arrival times, you'd do something like this:

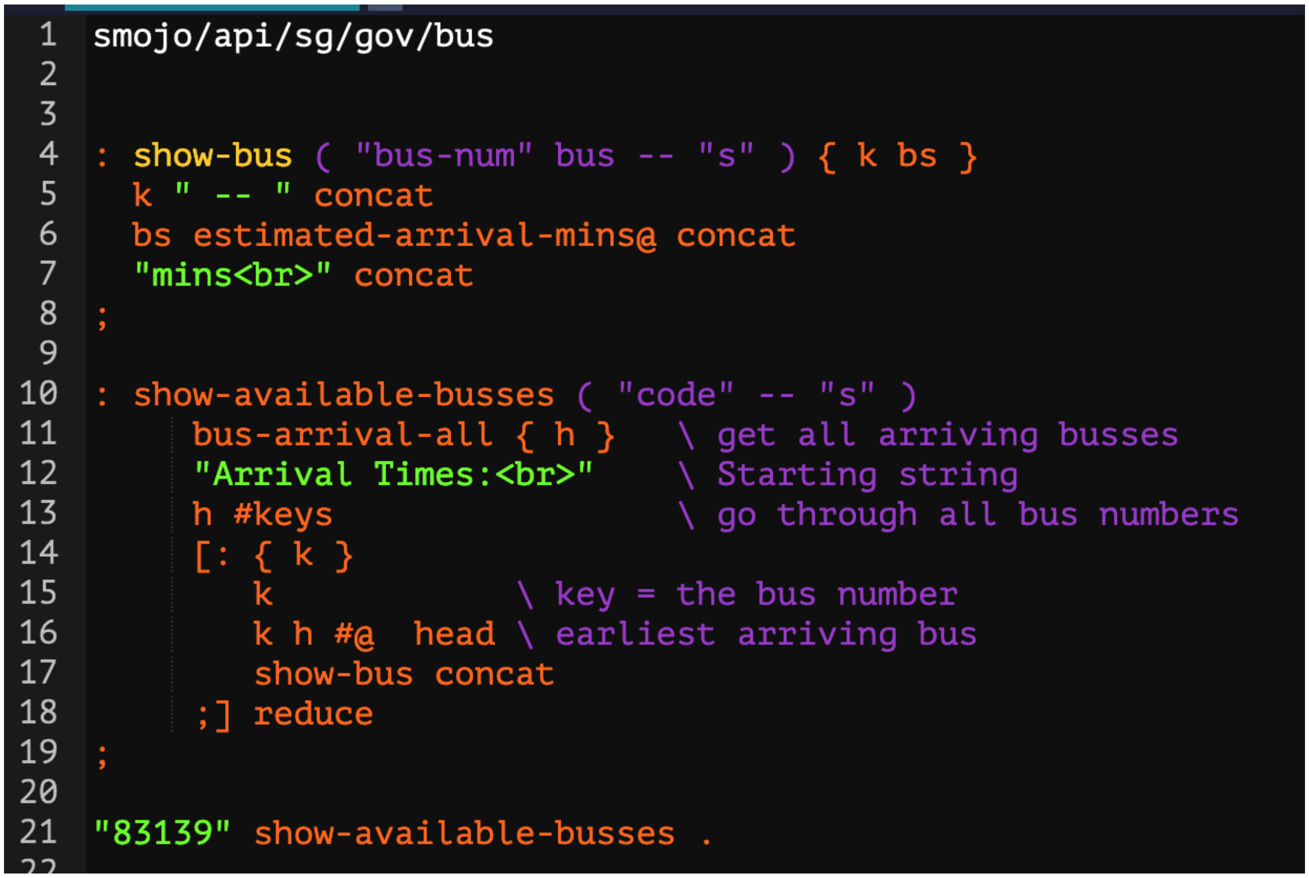
```
: show-bus ( "bus-num" bus -- "s" ) { k bs }
  k " -- " concat
  bs estimated-arrival-mins@ concat
  " mins<br>" concat
;

: show-available-busses ( "code" -- "s" )
     bus-arrival-all { h }    \ get all arriving busses
     "Arrival Times:<br>"     \ Starting string
     h #keys                  \ go through all bus numbers
     [: { k }
        k             \ key = the bus number
        k h #@  head \ earliest arriving bus
        show-bus concat
     ;] reduce
;
```

Listing 1: A first draft of **`show-available-busses`**

Exercise 2: Test the code above by copying and running it with a busstop code "83139". Remember to import `smojo/api/sg/gov/bus` first. How will you call **`show-available-busses`** ? **Hint**: take a careful look at the input and output of this word.

Let's go through this line by line:



```
 1  smojo/api/sg/gov/bus
 2
 3
 4  : show-bus ( "bus-num" bus -- "s" ) { k bs }
 5    k " -- " concat
 6    bs estimated-arrival-mins@ concat
 7    "mins<br>" concat
 8  ;
 9
10  : show-available-busses ( "code" -- "s" )
11      bus-arrival-all { h }   \ get all arriving busses
12      "Arrival Times:<br>"    \ Starting string
13      h #keys                 \ go through all bus numbers
14      [: { k }
15         k            \ key = the bus number
16         k h #@  head \ earliest arriving bus
17         show-bus concat
18      ;] reduce
19  ;
20
21  "83139" show-available-busses .
22
```

Figure 4: The first draft of show-available-busses

Lines 4 - 8 define a word to display a single bus.

Lines 10 - 19 are the main `show-available-busses` code.

Line 11 gets the hash of arriving busses. This is stored in a local called `h` .

Line 13 retrieves all keys - which are the bus numbers.

Lines 14 - 18 extract the bus number and earliest arriving bus and prints it out.

Pro TIP: It's OK if you don't understand the exact details of what's going on. You should however, get a rough overview and -- most importantly -- be able to run Listing 1 code successfully examples by yourself. **You must complete all Exercises**. Detailed understanding will come gradually.

Reductions

A common practice in Smojo is to do a **reduction** over a list/sequence. For example, suppose you have a list:

`{{ "dog" "cat" "bird" "horse" }}`

How would you print out every element in the list? You know that the period `.` prints out one item, but how about every item in this list? That's where reductions comes in.

You want to `reduce` the list using the period `.` word:

`{{ "dog" "cat" "bird" "horse" }} ' . reduce`

Pay careful note to the tick `'` word. Tick suppresses the action of the word after it, converting it into its XT. This XT is applied by reduce to every element of the list.

Exercise 3: Key in this reduction and click "Run" to see the result. Add new items and see if it works. What happens if you have the empty list `nil` as input? What if you mix numbers and strings?

`reduce ( seq xt -- * )` takes in a sequence and an XT and applies it to every element of the sequence.

What XT would you feed reduce so that each item is printed on a new line? One method might be to create a new word `print ( "s" -- )` that did this and get it's XT using tick:

```
: print ( "s" -- ) . cr ;

{{ "dog" "cat" "bird" "horse" }}  ' print  reduce
```

Exercise 4: Key this in and ensure it works. Change `print` so each line says for example "dog is an animal" , "cat is an animal" etc.

The word `print` is only needed as input into `reduce` . **In fact, we don't need the name `print` . All we need is its XT**. Recognizing this, we can just create an XT without being attached to a name:

```
{{ "dog" "cat" "bird" "horse" }}  :>  . cr  ;  reduce
```

Exercise 5: Key this in and ensure it works. Extend this example so that each line prints "dog is an animal" etc. **Hint**: You can use multiple lines.

The code `:> . cr ;` is called a quotation. When the program is run, it creates a nameless XT whose body has the action `. cr`

1. `:>` is like `:` but does not need a name.
2. `;` as usual completes the word.

Exercise 6: Look again at Figure 4's lines 13 - 18. What are the inputs of `reduce` on line 18? Can you identify the sequence and the XT fed into `reduce` ? **Hint #1**: what does #keys do? **Hint #2**: The words `[:` and `;]` are just like `:>` and `;` except that they work in compilation mode.

Pro TIP: You should understand this section Reductions as much as possible. You **must** complete all the Exercises in this section.

Busby V1

We are now ready to proceed to v1.

Step 1: Comment out the include for `busby-v0.m` and **uncomment** that for `busby-v1.m` . Ensure that just `busby-v1.m` is included - the other busby versions should not be included. They should be commented out.

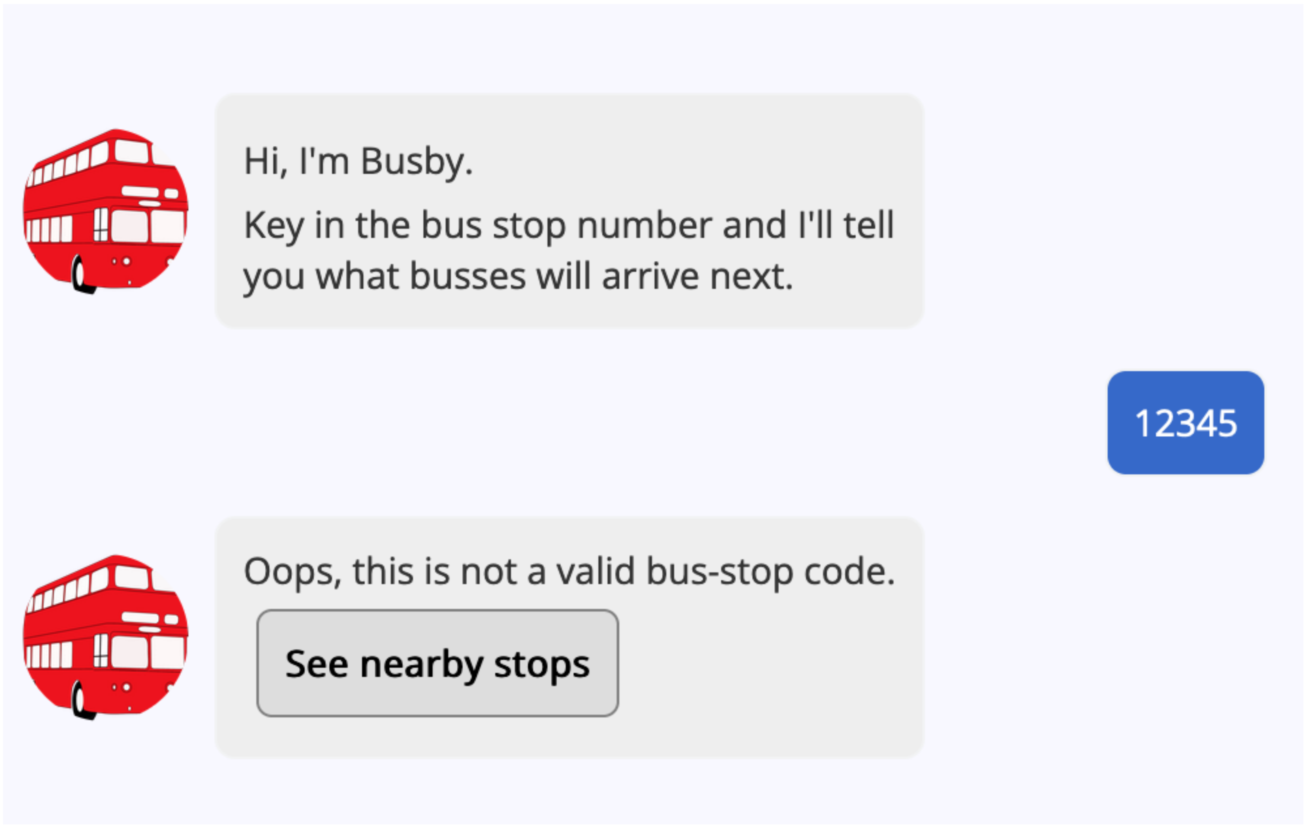
Step 2: `publish` and test out the app. What is the new feature introduced in this version?

Understanding Busby v1

V1 introduces two new features:

1. Check for a valid bus stop,
2. Display of nearest bus stops. This means that the bot knows where the user is in Singapore.

We'll examine this step by step. Enter in "12345" which is a 5-digit number but is not a valid bus-stop code. Here's what you should see:


Hi, I'm Busby.
Key in the bus stop number and I'll tell you what busses will arrive next.

12345

Oops, this is not a valid bus-stop code.

Figure 5: Busby responding to an invalid bus-stop code. Which template handles this?

Exercise 7a: Look at the code of `busby-v1.m` and identify which word is used to check the validity of the bus-stop code. **Hint**: you can guess which word it it by looking at the template that gives this response.

Exercise 7b: What is the API word used to verify the bus-stop code? Which cloud module is this word in?

Exercise 8: Get Busby v1 to give an IDK and click on "See nearby stops". You need to give Busby the permission to see your current location. Are the nearby bus stops listed correctly? If so, click on any of the displayed buttons. Do the bus arrivals appear? Which template handles this display?

Exercise 9: Identify the word used to display the closest bus-stops. **Hint**: you can see it from the template that is used to display the nearest stops.

Geolocation

Modern browsers have the ability to link up to GPS modules present in your laptop, computer or handphone. This is called **geolocation**. With it, you can tell where the user is (latitude and longitude) anywhere on earth. Of course, you need to set permissions on your phone or laptop.

In the chatbot engine, you can **enable** geolocation with the word **`\location`** placed anywhere in your code, outside of a word (ie, in interpretation mode).

Exercise 10: Where is geolocation enabled in Busby v1? Which line?

NOTE: Geolocation has nothing to do with APIs. You can use geolocation without using any APIs.

To read the user's current location, you use the word **`location`** in your Smojo words or templates. The word `location` takes no input, and its output depends on whether the user's location could be read.

1. If the user's location **can** be read, then the output is: `latitude` `longitude` **`false`** . ie, 3 items are placed on the stack, with **`false`** being on top.
2. If it cannot be read, then the output is `"ERROR MESSAGE"` **`true`** . Ie, 2 items only are placed on the stack. The top of the stack is **`true`** . The error message is a short string.

Exercise 11: What is the error message displayed to the user if geolocation fails? Hint: find where `location` is used and see what action is taken if the top of the stack is `true` .

Exercise 12a: How many nearest bus stops are displayed? Change the default number to 5, publish your bot and ensure your change works.

Exercise 12b: What API word is used to locate the nearest bus stops? On which cloud module does this word reside?

Transforming a Sequence

Another common action in Smojo is to transform one sequence into another. For example, suppose you want to transform the sequence:

```
{{ "dog" "cat" "bird" "horse" }}
```

to

```
{{ "DOG" "CAT" "BIRD" "HORSE" }}
```

How would you do this if you know that the word `ucase ( "s" -- "S" )` makes strings upper-cased? You can't use `reduce` , since `reduce` does not transform, it just applies the word.

`map ( seq xt -- seq )` does this job. It takes a sequence and and XT and applies the XT to every element of the sequence, transforming it. The output is a sequence of the same length. So,

```
{{ "dog" "cat" "bird" "horse" }} ' ucase map
```

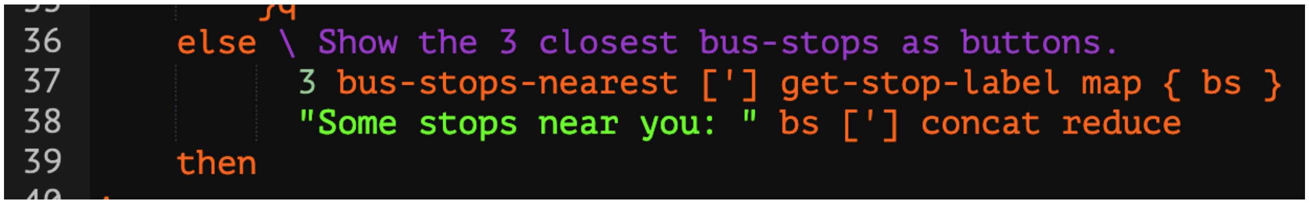
results in a transformed sequence, equivalent to `{{ "DOG" "CAT" "BIRD" "HORSE" }}`

Our old friend tick, `'` gets the XT of `ucase` to feed into `map` .

Exercise 13: How would you print out the mapped/transformed sequence to confirm that this is indeed the case? **Hint**: use a reduction.

Exercise 14: How would you transform `{{ "dog" "cat" "bird" "horse" }}` so that it becomes `{{ "dogs" "cats" "birds" "horses" }}` ? **Hint**: `concat ( "a" "b" -- "ab" )` concatenates two strings. Be sure to prove your code works using a reduction that prints it out.

As you can see from Exercises 13 and 14, it is a common thing to take a sequence, transform it using `map` then perform a `reduce` to get a single result. You can see this happening in `busby-v1.m` 's `closest-bus-stops` word:



```
    else \ Show the 3 closest bus-stops as buttons.
        3 bus-stops-nearest ['] get-stop-label map { bs }
        "Some stops near you: " bs ['] concat reduce
    then
```

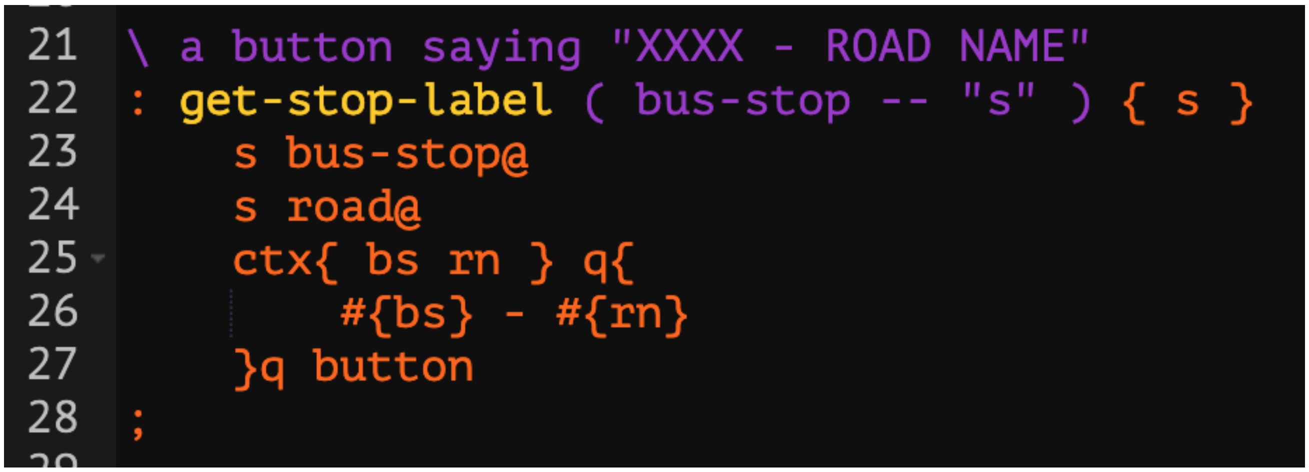
Figure 6: a Map - Reduction in Busby v1.

Exercise 15: Be sure you understand clearly what the `map` in line 37 and the reduction in line 38 of Figure 6 are doing. Specifically, answer these questions:

1. What are the sequence and XT which are inputs of `map` in line 37?
2. What are the sequence and XT which are inputs of `reduce` in line 38?

Displaying Bus Stops

Take a look at the word `get-stop-label` of line 37 in Figure 6:



```
\ a button saying "XXXX - ROAD NAME"
: get-stop-label ( bus-stop -- "s" ) { s }
    s bus-stop@
    s road@
    ctx{ bs rn } q{
        #{bs} - #{rn}
    }q button
;
```

Figure 7: get-stop-label

Exercise 16: This word takes a `bus-stop` object as input and outputs a string (a chatbot button, in fact). Where do the words `bus-stop@` and `road@` come from? What do they do? Use this knowledge to change `get-stop-label` so that it prints the **description** of the bus stop, instead of the road name.

Busby v2

This last version of Busby is by far the most interesting. In this version we will change `closest-bus-stops` so that it not only displays the buttons of nearest bus-stops, but also shows a map of the surrounding area with the stops plotted in.

Step 1: As usual, comment out other busby's on `main.m` and ensure `busby-v2.m` is uncommented.

Step 2: publish and play with the app. Specifically test out the "See nearby stops" function. Are the nearby bus-stops displayed correctly on a geographical map?

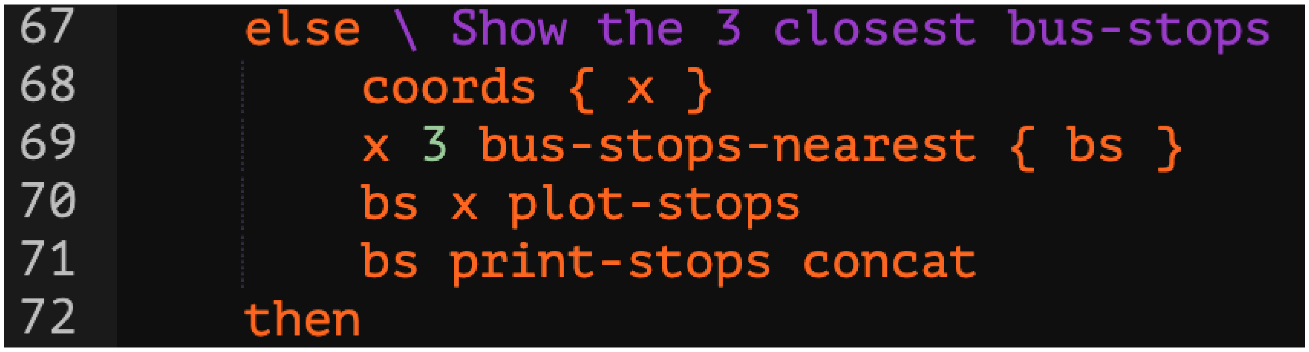
Understanding Busby v2

The new API we've used is `smojo/api/sg/gov/map` , which has 2 clusters of functions:

1. `sg-map` creates a blank map, `+point` adds a locator and `plot-map` gives you a url that you can insert into `image` or `(image)` - if you need to customize the CSS. `color` is a helper word to define new colors for `+point` . Be sure to use `smojo/doc` to view the documentation for these words.
2. `address-search` finds an address given a part of its name. For example, "ista" will locate Istana and many other buildings whose name contain "ista". `rev-geo` is a reverse geographical search, and will find buildings within a given radius of a coordinate.

Exercise 17: Open up `busby-v2.m` . Which of these clusters is used in v2?

The change to the code to plot the map is in `closest-bus-stops` :



```
67       else \ Show the 3 closest bus-stops
68           coords { x }
69           x 3 bus-stops-nearest { bs }
70           bs x plot-stops
71           bs print-stops concat
72       then
```

Figure 8: The amend part of closest-bus-stops

Let's examine this line by line:

Line 68 transforms the latitude and longitude output from the `location` word into a single `coord` object, which is then named `x` . So, `x` is where the user is, based on geolocation.

Line 69 gets the 3 nearest bus-stops from `x` . This result is called `bs` , short for busstops.

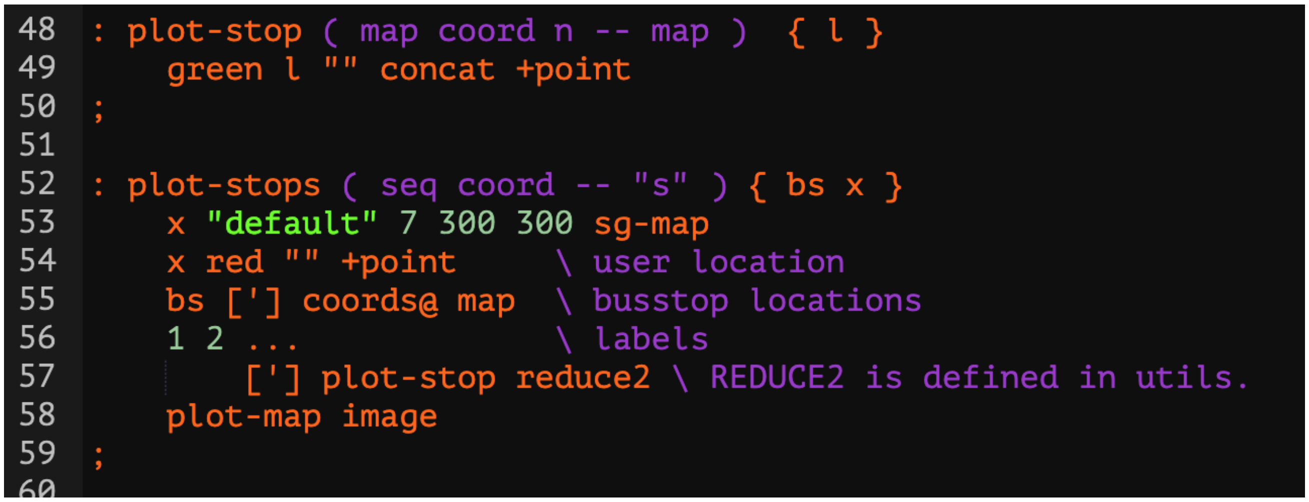
Line 70 plots out the stops on a geographical map. The word used is `plot-stops` .

Line 71 displays the buttons. The word used is `print-stops` .

Let's examine these two words, `plot-stops` and `print-stops` in more detail.

Plotting the Map

Take a close look at plot-stops and plot-stop:



```
: plot-stop ( map coord n -- map )  { l }
   green l "" concat +point
;

: plot-stops ( seq coord -- "s" ) { bs x }
   x "default" 7 300 300 sg-map
   x red "" +point      \ user location
   bs ['] coords@ map   \ busstop locations
   1 2 ...              \ labels
      ['] plot-stop reduce2 \ REDUCE2 is defined in utils.
   plot-map image
;
```

Figure 9: plot-stops and plot-stop

`plot-stops` takes in a central coordinate `x` , and a sequence of bus-stops `bs` , to plot. The coordinate `x` is used to define a small 300x300 pixel map, at zoom-level 7 with the `default` color scheme (see the documentation for `sg-map` for details on these and all available options).

Exercise 18: Change the size of the map to 350 x 300 and select a different color scheme. Publish the bot and test your results.

The resulting blank map is left on the stack. Subsequently, we want to plot in the user's location `x` with a `red` marker using `+point` .

Exercise 19: Change the user's marker to a blue one. **Hint**: you have to define `blue` and use it.

Once that is done, the bus stops can be plotted. The mapping API only allows a single character to be displayed on any map:

Figure 10: A single marker on a geographical map. Only single characters are allowed (or nothing at all) by the API. In this example the character displayed is "2".

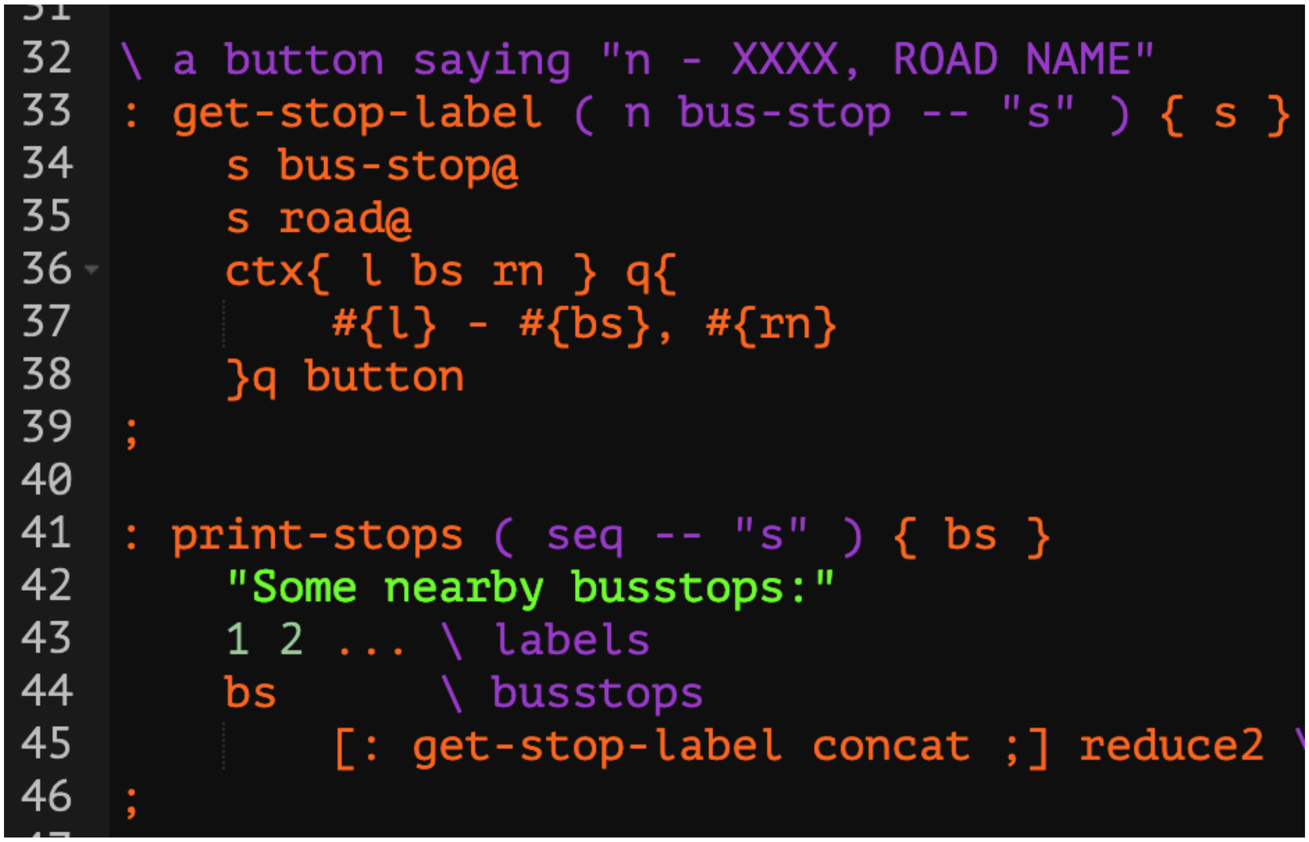
So, we have to come up with numbered labels 1 2 ... etc to be shown on the markers. That's what lines 55, 56 and 57 do:

1. **Line 55** transforms a sequence of `bus-sop` objects into a sequence of coordinates, using `map` . `coords@` extracts the coordinate object from a `bus-stop` object.
2. **Line 56** creates a list of numbers 1 2 ... This is an infinite sequence. These from the marker labels.
3. **Line 57** takes these two sequences (coordinates and labels) and runs a reduction over them. Smojo does not have a built-in word to reduce **two** sequences at once, so I have made one for you in `utils.m` , called `reduce2` . **`reduce2`** `( seqA seqB xt -- * )` takes two sequences and an XT for the reduction. The XT used is `plot-stop` .
4. Finally, **Line 58** plots the geographical map using `plot-map` and uses `image` to create the image HTML tag.

Exercise 20: Edit the code to plot up to 9 nearest stops. Test out your bot after making these changes. What happens if you exceed 9? Should this limitation concern us? Can you think of a method to overcome it?

Displaying the Buttons

Displaying the buttons for the nearest stops follows a similar pattern. print-stops does this and its input is the sequence of nearby bus-stops.



```
32  \ a button saying "n - XXXX, ROAD NAME"
33  : get-stop-label ( n bus-stop -- "s" ) { s }
34      s bus-stop@
35      s road@
36      ctx{ l bs rn } q{
37          #{l} - #{bs}, #{rn}
38      }q button
39  ;
40
41  : print-stops ( seq -- "s" ) { bs }
42      "Some nearby busstops:"
43      1 2 ... \ labels
44      bs      \ busstops
45          [: get-stop-label concat ;] reduce2 \
46  ;
```

Figure 11: print-stops and get-stop-label

`get-stop-label` is a **helper function** that converts its input (the label and `bus-stop` object) into a string (in this case a chatbot button).

Lines 43 - 45 follow the same pattern as before:

1. `1 2 ...` is an infinite sequence on numbers for marker labels.
2. `bs` is the sequence of nearby bus-stops.
3. `reduce2` is used to reduce these sequences using `get-stop-label` . `concat` is used to concatenate the button to the "Some nearby busstops:".

Pro TIP: At this point, it's OK to have a rough understanding of how `print-stops` works. But you need to be really clear how `get-stop-label` works.

Congratulations! You made it to the end... well, almost to the end. You should now try to complete the homework.

Homework

Q1: Add in a "Refresh" button to v0 so that the user can click it to refresh the arrival times, instead of having to type in the bus-stop code.

Q2: Add in a "Save" button so that the user can save the last stop. This is remembered after he closes his chatbot. When he opens it again, the bot's welcome message is amended, so he can type "f" or "favourite" to retrieve and display the bus arrivals for this stop. Hint: You need to use persistent storage (Video CU101- Session 8).

Q3: Using v1, display the distance (in metres) of a bus-stop to the user's location. **Hint**: You need to **approximately** convert degrees to metres. Write a word for this.

Q4: Add a new microtopic for the user to explore bus routes. This topic should display all stops for a given bus number. Just plot these on a map.

Q5: The user knows that Bus 174 is the one he needs to catch. But which stop is closest to his current location? Write a microtopic to help him with this. It should (a) get the nearest stop to him for any bus number. You can show him the 3 nearest stops and plot it on a geographical map. **Hint 1**: you need to use `smojo/api/sg/gov/bus/info` 's `bus-route` word and search through that list. **Hint 2**: You can use `#sort-by-values ( # -- # )` to get a new hash sorted by its values. Retrieve the keys using `#keys` . The keys can be bus-stop objects and values can be distance from the user. **Hint 3**: To get the first n items from a sequence, use `take ( seq n -- seq )` .

Q6*: Explore the `smojo/api/sg/gov/places` API. Create a microtopic to help the user explore F&B places. This should help him find suitable food places and suggest busses he might use to get there. Assume he's only interested in direct busses from where he is. If there is no direct bus, the bot should say "Oops, no direct bus is available".

* * *