Custom JS-transitions

while custom css generally take care of everything, there are effects that can't be acheived without JS:

Ex → typewriter effect

```
function typewriter (node, {speed = 1 }) {
    const valid = node.childNodes.length === 1 && node.childNodes[0].nodeType === Node.TEXT_NODE
    if(!valid) throw error
    const text = node.textContent;
    const duration = text.length/(speed * 0.01);
    return {
        duration,
        tick: (t) => {
            const i = Math.trunc(text.length * t); // it gives the % of length of text available a/c to t.
            node.textContent = text.slice(0,i);
        }
    };
}
```

The tick f^n is called repeatedly from beginning of transition to the end with value of t, inc. from 0 to 1.

We can ~~[illegible]~~ track start and end of transitions because transitions provide events to do so.
On the comp. that has transition, applying event catchers like on:introstart = call back f^n to do stuff like printing status

on:outrostart = "

~~[illegible]~~

on:introend = "

on:outroend = "

Global Transitions

ordinarily transitions are transitions are triggered when when an entire block is added or removed from DOM. Transitions being applied to handle adding or removing individual items from list or component would affect only adding/removing individual item not on entire list. If we make the entire list appear or disappear, transition effect won't be observed but if add the term global to it, the transition effect is applied to the entire list as well.
The global modifier extends the scope of transition. Normally, transitions are local.

Key blocks

Key blocks destroy and recreate their contents when the value of an expression changes. This is useful if you want an element to play its transition whenever a value changes instead of only when the element enters or leaves the DOM.
Here, for example, we'd like to play the typewriter transition from transition.js whenever the loading message i.e., i changes. Wrap the <p> element in a key block.

```
{#key i}
  <p in:typewriter={{speed:10}}>
     {message[i]||''}
  </p>
{/key}
```

If we would not have used key blocks, the typewriter effect would have invalid as DOM is not loading/unloading but rather just changing values.

Deferred transitions

Deferred transitions allow for the coordination of transitions b/w multiple elements. In simple terms, instead of a simple transition that affects just one element, deferred transitions enable elements to communicate and transition smoothly as they move b/w diff. state or list. The key idea is to make the elements "wait" until the counterpart is ready to transition.

In the example, crossfade transition was used to showcase the effect of movement of a todo from one list to the other.

* Crossfade has 2 transitions :-

↳ send → used when the element is sent or removed from its original locⁿ.

↳ receive → used when the element is received or added to its new location.

We customized these fns using options provided to make it suitable for our example. We added fall fn to it that will run in case parameter (id of todo) is not provided.

(In the example todo store was a writable store with several custom fns for handling functionalities of adding a todo, deleting a $\underset{\downarrow}{\boxed{\text{this}}}$ todo, or marking as done or undone.

Animations

The animate directive is responsible for providing motion to elements. One such animation fn is flip → first, last, invert, play.

on the element that needs to be animated,
apply → animate: flip

Actions

Actions are essentially element-level lifecycle functions.
Useful in:
- ↳ ~~[illegible]~~ interfacing with 3rd party libraries
- ↳ lazy-loaded images
- ↳ tooltips
- ↳ adding custom event handlers

Trapfocus

Trapfocus action "trap" the keyboard focus inside the model so that when user navigates through the options using the Tab key, focus stays within the model instead of going outside it. Models (popups) main function is to gather some input from user by focusing on the current available options. But this feature is not added by default in it.

Actions are functions that lets us hook into lifecycle of an element. They allow us to manipulate the element directly. When an element is created and inserted into the DOM, Svelte will call the action fn with that element. This action can return a destroy fn to clean up when the element is removed.
In the given example, in actions.js, we first tried to add focus to first focusable element like button or input. Then we added an event listener inside the

inside. It checks if user have pressed the 'tab' key or not. It is to keep the focus inside the model node itself. Then we created a destroy fn that will destroy the event handler. and focus is returned to wherever it was before the model was opened (which is important for accessibility). This destroy fn will be called the DOM element would be unmounted. Actions are purely defined as a fn (prefferably in a separate file like stores) that takes the node itself modifies the node as required and return destroy fn. Then the element that need to implement it would apply use: action-name. Then the fn (action fn) would be called first with the element associated.

Adding parameters to actions :

Like transitions and animations an action can take an argument which the action fn will be called with alongside the elements it belongs to.

In the given example, we were trying add tooltip action (tippy) from tippy.js (tooltip, mouseover and any other stuffs are predefined in this library, It's cool). We added use: tooltip to the button over which we had to add tooltip. But it did not return any effect. It shows empty tooltip. It's because it requires some parameters (tippy needs it) apart from node. It takes options object as input. We can send content and theme to it.

Lets say we need to show the tooltip content according to some input by the user. In that case updated value won't show up in the tool-tip. So, in order to handle updates, we can add a fn update (it will be returned with destroy fn). This update fn would take options as input and set Props of the tooltip (Basic updating