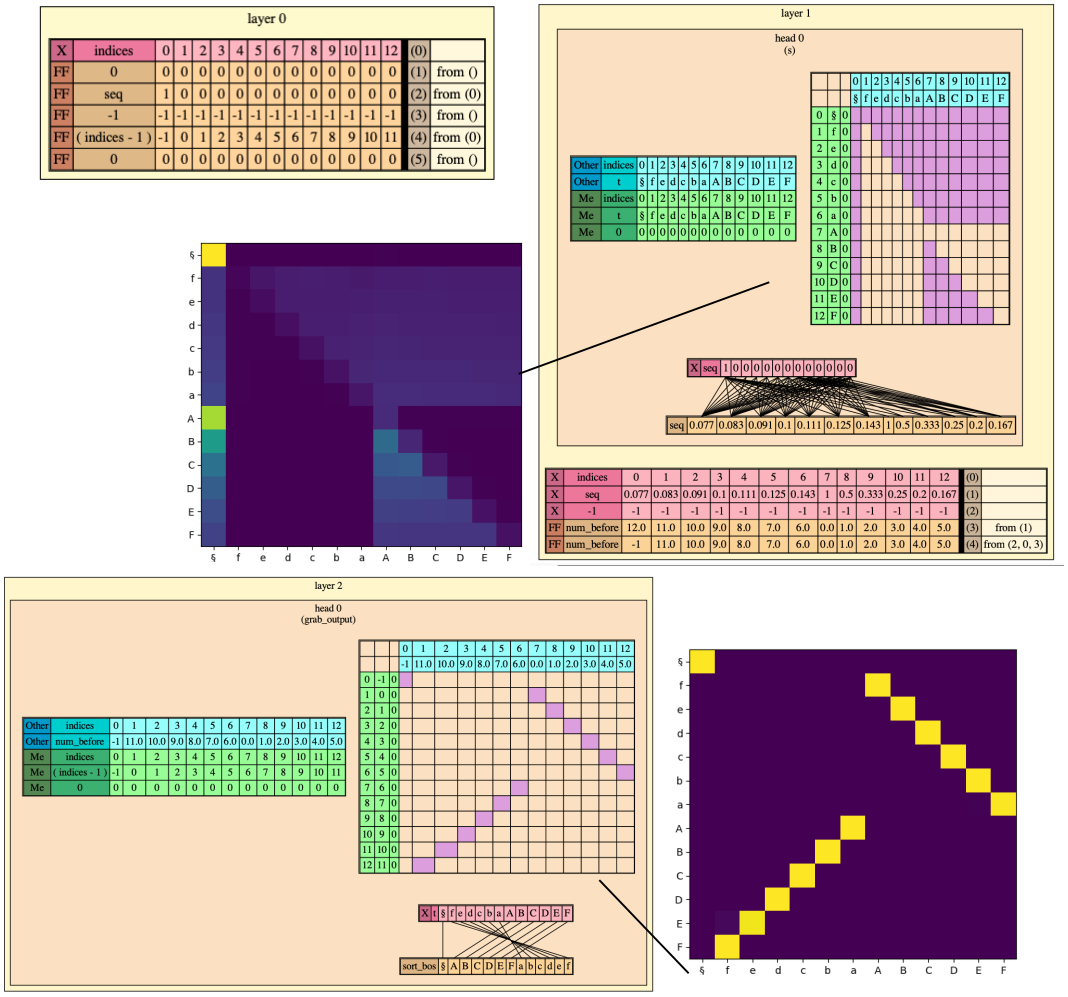

Figure 6: Computation flow in compiled architecture from RASP solution for sort (with BOS token), alongside heatmaps from the corresponding heads in a transformer trained with both target and attention supervision on the same task and RASP solution. The RASP solution is simply written `sort(tokens,tokens,assume_bos=True)`, using the function `sort` shown in Figure 15. Both the RASP architecture and the transformer are applied to the input sequence `"§fedcbaABCDEF"`.

Appendices

In Appendix A we give training details from the experiments in this paper, as well as additional results from the transformers trained to mimic RASP-predicted attention patterns. The exact RASP solutions for all tasks considered in the paper, as well as an implementation of the operation `selector_width` in terms of other operations (which have direct translation to a transformer), are presented in Appendix B. This section also presents the computation flows in compiled architectures for several of these solutions.

Note. References to appendices D and E in the submitted draft should in fact be to B and A, respectively. Our apologies.

A. Experiments

A.1. Results: Attention-regularised transformers

We trained 3 transformers with a target attention pattern according to our RASP solutions, these 3 being for the tasks double-histogram, sort, and most-freq as described in the paper. All of these reached high ($99{+}\%$) accuracy on their sequence-to-sequence task, computed as fraction of output tokens predicted correctly. Plotting their attention patterns also shows clear similarity to those of the compiled RASP programs:

For the *double-histogram* task, a full compiled architecture is presented on the sequence §aabbaa in Figure 17. Additionally, in Figure 1, just its attention patterns are pre-

sented alongside the corresponding attention heads from its attention-regularised transformer, this time both on the sequence `§aabbaabb`.

For the *sorting* task, we present a full computation flow on the input sequence `§fedcbaABCDEF`, alongside the corresponding attention heads of the regularised transformer on the same sequence, in Figure 6. The regularised transformer had input alphabet of size 52 and reached test accuracy 99.0% on the task (measured as percentage of output positions where the correct output token had the maximum probability).

For the *most-freq* task (returning each unique token in the input, by descending order of frequency, and padding the rest with the BOS token) we do the again show a computation flow alongside the regularised transformer, this time in Figure 7 and with the sequence `§aabbcddd`. On this task the regularised transformer had input alphabet of size 26 and reached test accuracy 99.9%.

A.2. Training Details

In the upper bound and tightness experiments (Section 5), for each task and layer/head specification, we train transformers with embedding dimension 256 and feed-forward dimension 512 on the task for 100 epochs. We use learning rates 0.0003 and 0.0001, and learning rate decay $\gamma = 0.98$ and 0.99, training 4 transformers overall for each task. We use the ADAM optimiser and no dropout. Each transformer is trained on sequences of length $0--100$, with train/validation/test set sizes of $50,000$, $1,000$, and $1,000$ respectively. Excluding the BOS token, the alphabet sizes are: 3 and 5 and for Dyck-1 and Dyck-2 (the parentheses, plus one neutral token), 100 for reverse and sort, and 26 for the rest (to allow for sufficient repetition of tokens in the input sequences). All input sequences are sampled uniformly from the input alphabet and length, with exception of the Dyck languages, for which they are generated with a bias towards legal prefixes to avoid most outputs being `F`.

For the attention regularised transformers, we make the following changes: first, we only train one transformer per language, with learning rate 0.0003 and decay 0.98. We train each transformer for 250 epochs (though they reach high validation accuracy much earlier than that). The loss this time is added to an MSE-loss component, computed from the differences between each attention distribution and its expected pattern. As this loss is quite small, we scale it by a factor of 100 before adding it to the standard output loss.

B. RASP programs and computation flows for the tasks considered

B.1. `selector_width`

The RASP implementation of `selector_width` is presented in Figure 9. The core observation is that, by using a selector that always focuses on zero (`or0` in the presented code), we can compute the inverse of that selector's width by aggregating a 1 from position 0 and 0 from everywhere else. It then remains only to make a correction according to whether or not the selector was actually focused on 0, using the second selector `and0` (if there isn't a beginning-of-sequence token) or our prior knowledge about the input (if there is).

B.2. RASP solutions for the paper tasks

We now present the RASP solutions for each of the tasks considered in the paper, as well as an implementation of the RASP primitive `selector_width` in terms of only the primitives `select` and `aggregate`.

The solution for histograms, with or without a BOS token, is given in Figure 11. The code for double-histograms (e.g., `hist2("aaabbccdef")=[1,1,1,2,2,2,2,3,3,3]`) is given in Figure 12. The general sorting algorithm (sorting any one sequence by the values ('keys') of any other sequence) is given in Figure 13, and sorting the tokens by their frequency ("Most freq") is given in Figure 14. Descriptions of these solutions are in their captions.

The Dyck-PTF Languages *Dyck-1-PTF* First each position attends to all previous positions up to and including itself in order to compute the balance between opening and closing braces up to itself, not yet considering the internal ordering of these. Next, each position again attends to all previous positions, this time to see if the ordering was problematic at some point (i.e., there was a negative balance). From there it is possible to infer for each prefix whether it is balanced (T), could be balanced with some more closing parentheses (P), or can no longer be balanced (F). We present the code in Figure 15.

Dyck-2-PTF For this descripition we differentiate between instances of an opening and closing parenthesis (*opener* and *closer*) matching each other with respect to their position within a given sequence, e.g. as `(,>` and `{,]` do in the sequence `({]>`, and of the actual tokens matching with respect to the pair definitions, e.g. as the token pairs `{,}` and `(,)` are defined. For clarity, we refer to these as structure-match and pair-match, respectively.

For a Dyck-n sequence to be balanced, it must satisfy the balance checks as described in Dyck-1 (when treating all openers and all closers as the same), and additionally, it must satisfy that every structure-matched pair is also a pair-

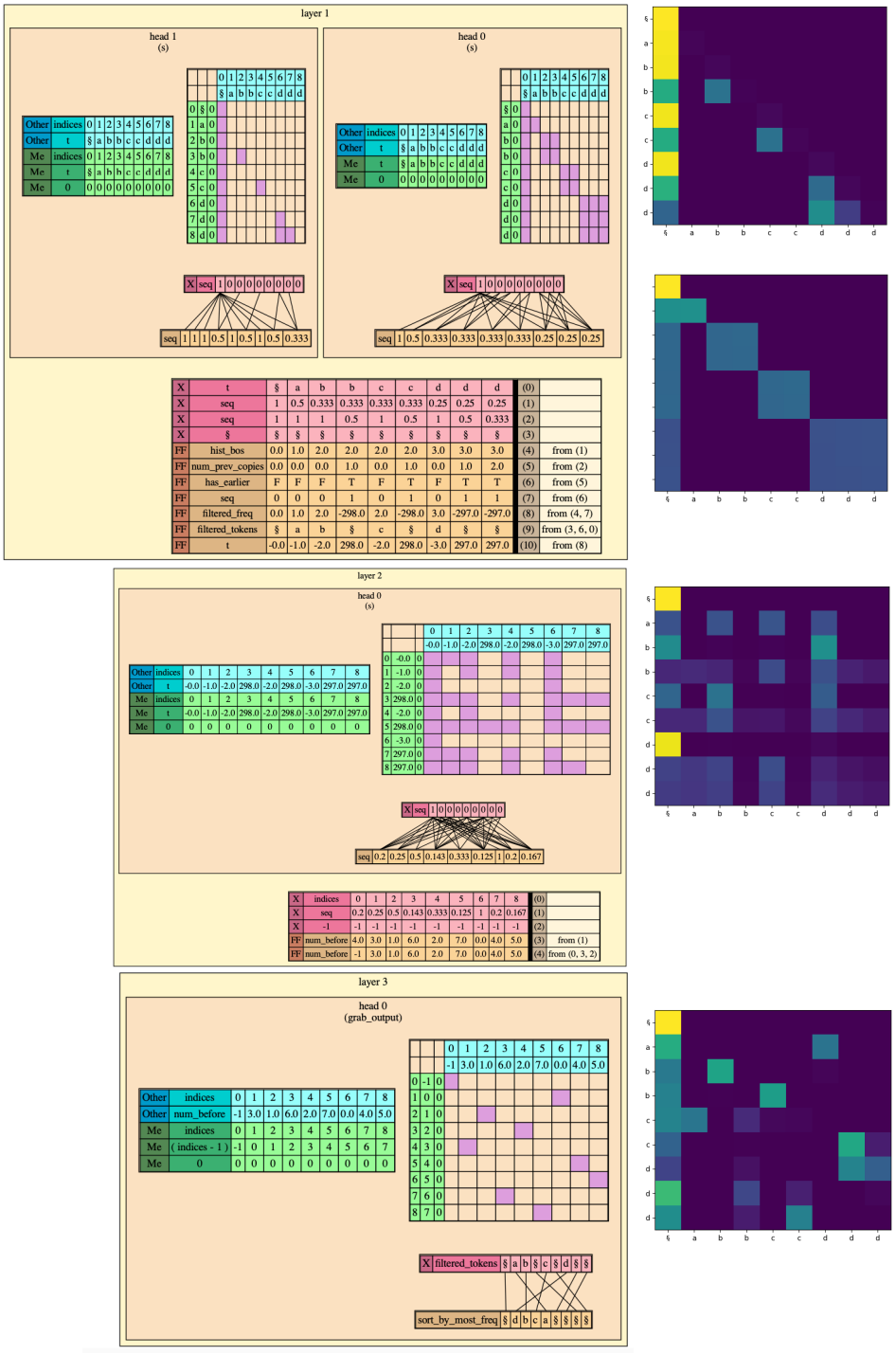

Figure 7: Computation flow in compiled architecture from RASP solution for sorting by frequency (returning all unique tokens in an input sequence, sorted by decreasing frequency), alongside heatmaps from attention heads in transformer trained on same task and regularised to create same attention patterns. Both are presented on the input sequence §abbccddd, for which the correct output is §dbca. The transformer architecture has 3 layers with 2 heads apiece, but the RASP architecture requires only 1 head for each of the second and third layers. We regularised only one for each of these and present just that head.

```
pairs = ["()","{}","[]"]; # etc ...
openers = [p[0] for p in pairs];
closers = [p[1] for p in pairs];
opens = tokens in openers;
closes = tokens in closers;
n_opens = num_prevs(opens);
n_closes = num_prevs(closes);

depth = n_opens - n_closes;
delay_closer =
    depth + indicator(closes);
earlier_same_depth =
    select(delay_closer,delay_closer,==)
        and
    select(indices,indices,<=);
depth_index =
    selector_width(earlier_same_depth);
open_for_close =
        select(opens,True,==) and
        select(delay_closer,
               delay_closer,==) and
        select(depth_index,
               depth_index-1,==);
matched_opener =
    aggregate(open_for_close,tokens,"-");
opener_matches = matched_opener+t in pairs;
mismatch = closes and not opener_matches;
had_problem =
    num_prevs(mismatch or depth<0 )>0;
return "F" if had_problem else
        ("T" if depth==0 else "P");
```

Figure 8: Pure RASP code (as opposed to with an additional select-best operation) for computing Dyck-3-PTF with the parentheses `(,)`, `{,}` and `[,]`. The code can be used for any Dyck-n by extending the list `pairs`, without introducing additional layers or heads.

match.

We begin by using the function `num_prevs` from Figure 15 to compute balances as for Dyck-1, ignoring which token pair each opener or closer belongs to. Next, we create an attention pattern `open_for_close` that focuses each closer on its structure-matched opener, and use that pattern to pull up the structure-matched opener for each closer (the behaviour of that pattern on closers that do not have structure-matched openers is not important: in this case there will anyway be a negative balance at that closer). For each location, we then check that it does not have an earlier negative balance, and it does not have an earlier closer whose structure-matched opener is not a pair-match. If it fails these conditions the output is F, otherwise it is T if the current balance is 0 and P otherwise. The remaining challenge is in computing `open_for_close`.

In pure RASP—i.e., within the language as presented in this work—this is realisable in two steps. First, we number each parenthesis according to how many previous parentheses have the same depth as itself, taking for openers the depth after their appearance and for closers the depth before. For example, for `(())()`, the depths are `[1,2,2,1,1,1]`, and the depth-index is `[1,1,2,2,3,3]`. Then, each closer's structure-matched opener is the opener with the same depth as itself, and depth-number immediately preceding its own. This solution is given in Figure 8, and compiles to 4 layers with maximum width 2.

However, by adding the theoretical operation `select_best`, and a `scorer` object similar to selectors (with numbered values as opposed to booleans), we can simplify the computation of `open_for_close` to simply: the last opener with the same depth as the closer's, that is still before the closer. This would be obtained as `select_best( select(adjusted_depth,adjusted_depth,==)` and `select(indices,indices,<) , score(indices,0,+))`. In this case, the depth-index of each position does not need to be computed in order to obtain `open_for_close`, saving the layer and 2 heads that its compilation creates.

B.3. Computation flows for select solutions

RASP can compile the the architecture of any s-op, and display it with an example input sequence. The command is `draw(s2s,inp)` where `s2s` is the target s-op and `inp` is the example sequence to display, e.g., `draw(dyck1,"(())")`.

Example computation flows for `hist_bos` and `reverse` are given in the main paper in Figures 5 and 4, respectively.

An example computation flow for `hist_nobos` is given in Figure 16. The double-histogram flow partially shown in Figure 1 is shown in full in Figure 17. Computation flows for the compiled architectures of `sort` and for `most_freq` (as solved in Figures 13 and 14) are shown in full, alongside

```
def selector_width(sel,
             assume_bos = False) {

    light0 = indicator(
                 indices == 0);
    or0 = sel or select_eq(indices,0);
    and0 =sel and select_eq(indices,0);
    or0_0_frac =aggregate(or0, light0);
    or0_width = 1/or0_0_frac;
    and0_width =
        aggregate(and0,light0,0);

    # if has bos, remove bos from width
    # (doesn't count, even if chosen by
    #  sel) and return.
    bos_res = or0_width - 1;

    # else, remove 0-position from or0,
    # and re-add according to and0:
    nobos_res = bos_res + and0_width;

    return bos_res if assume_bos else
                 nobos_res;
}
```

Figure 9: Implementation of the powerful RASP operation `selector_width` in terms of other RASP operations. It is through this implementation that RASP compiles `selector_width` down to the transformer architecture.

the attention patterns of respectively attention-regularised transformers, in Appendix A. Computation flows for Dyck-1-PTF and Dyck-2-PTF are shown in Figure 18 and Figure 19.

```
reverse = aggregate(
    select(indices,
           length-indices-1,==)
    tokens );
```

Figure 10: RASP one-liner for reversing the original input sequence, `tokens`. This compiles to an architecture with two layers: `length` requires an attention head to compute, and `reverse` applies a `select-aggregate` pair that uses (among others) the s-op `length`.

```
def histf(seq, assume_bos = False) {
    same_tok = select(seq,seq,==);
    return selector_width(same_tok,
            assume_bos= assume_bos);
}
```

Figure 11: RASP program for computing histograms over any sequence, with or without a BOS token. Assuming a BOS token allows compilation to only one layer and one head, through the implementation of `selector_width` as in Figure 9. The `hist_bos` and `hist_nobos` tasks in this work are obtained through `histf(tokens)`, with or without `assume_bos` set to `True`.

```
def has_prev(seq) {
    prev_copy =
        select(seq,seq,==) and
        select(indices,indices,<=);
    return aggregate(prev_copy,1,0)>0;
}

is_repr = not has_prev(tokens);
same_count =
    select(hist_bos, hist_bos,==);
same_count_reprs = same_count and
    select(isnt_repr, False,==);
hist2 =selector_width(
        same_count_reprs,
        assume_bos = True);
```

Figure 12: RASP code for hist-2, making use of the previously computed `hist` s-op created in Figure 11. We assume there is a BOS token in the input, though we can remove that assumption by simply using `hist_nobos` and removing `assume_bos=True` from the call to `selector_width`. The segment defines and uses a simple function `has_prev` to compute whether a token already has an copy earlier in the sequence.

```
def sort(vals,keys,assume_bos=False) {
    smaller = select(keys,keys,<) or
        (select(keys,keys,==) and
         select(indices,indices,<) );
    num_smaller =
        selector_width(smaller,
            assume_bos=assume_bos);
    target_pos = num_smaller if
                    not assume_bos else
    (0 if indices==0 else (num_smaller+1));
    sel_new =
        select(target_pos,indices,==);
    sort = aggregate(sel_new,vals);
}
```

Figure 13: RASP code for sorting the s-op `vals` according to the order of the tokens in the s-op `keys`, with or without a BOS token. The idea is for every position to focus on all positions with keys smaller than its own (with input position as a tiebreaker), and then use `selector_width` to compute its target position from that. A further select-aggregate pair then moves each value in `val` to its target position. The sorting task considered in this work's experiments is implemented simply as `sort_input=sort(tokens,tokens)`.

```
max_len = 20000;
freq = hist(tokens,assume_bos=True);
is_repr = not has_prev(tokens);
keys = freq -
    indicator(not is_repr) * max_len;
values = tokens if is_repr else "§"
most_freq = sort(values,keys,
                 assume_bos=True);
```

Figure 14: RASP code for returning the unique tokens of the input sequence (with a BOS token), sorted by order of descending frequency (with padding for the remainder of the output sequence). The code uses the functions `hist` and `sort` defined in Figures 11 and 13, as well as the utility function `has_prev` defined in Figure 12. First, `hist` computes the frequency of each input token. Then, each input token with an earlier copy of the same token (e.g., the second `"a"` in `"baa"`) is marked as a duplicate. The key for each position is set as its token's frequency, minus the maximum expected input sequence length if it is marked as a duplicate. The value for each position is set to its token, unless that token is a duplicate in which case it is set to the non-token §. The values are then sorted by the keys, using `sort` as presented in Figure 13.

```
def num_prevs(bools) {
    prevs = select(indices,indices,<=);
    return (indices+1) *
           aggregate(prevs,
                     indicator(bools))
}
n_opens = num_prevs(tokens=="(");
n_closes = num_prevs(tokens==")");
balance = n_opens - n_closes;
prev_imbalances = num_prevs(balance<0);
dyck1PTF = "F" if prev_imbalances > 0
                    else
          ("T" if balance==0 else "P");
```

Figure 15: RASP code for computing Dyck-1-PTF with the parentheses `(` and `)`.

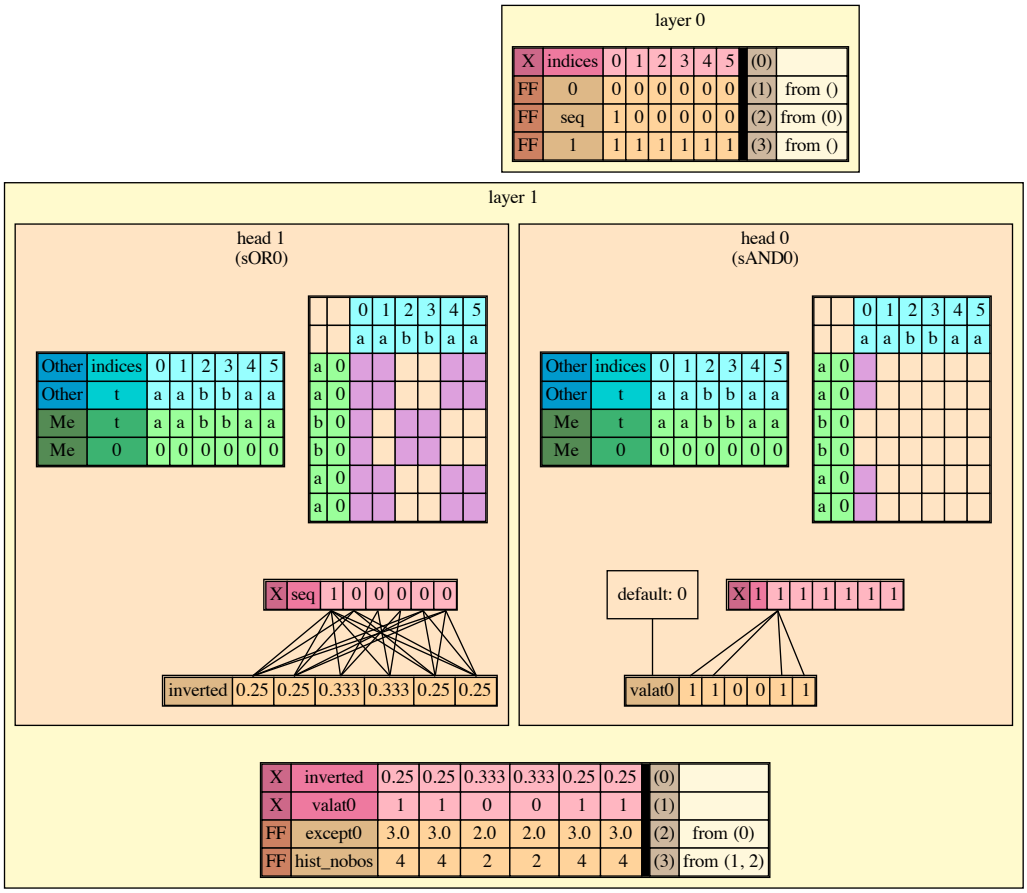

Figure 16: Computation flow in compiled architecture from RASP solution for histogram without a beginning-of-sequence token (using `histf(tokens)` with `histf` from Figure 11). We present the short sequence `"aabbaa"`, in which the counts of `a` and `b` are different.

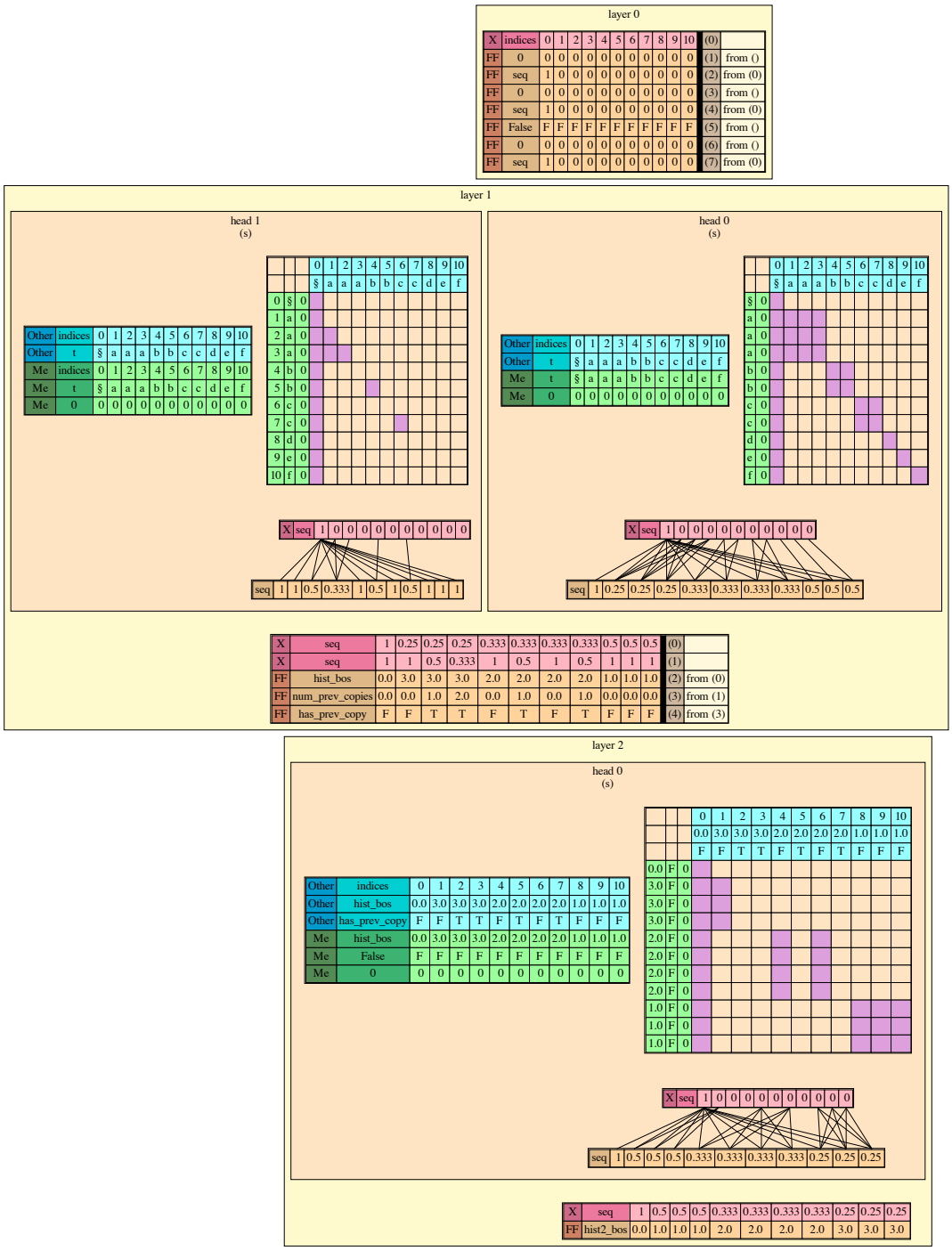

Figure 17: Computation flow in compiled architecture from RASP solution for double-histogram, for solution shown in Figure 12. Applied to "§aaabbccdef", as in Figure 1.

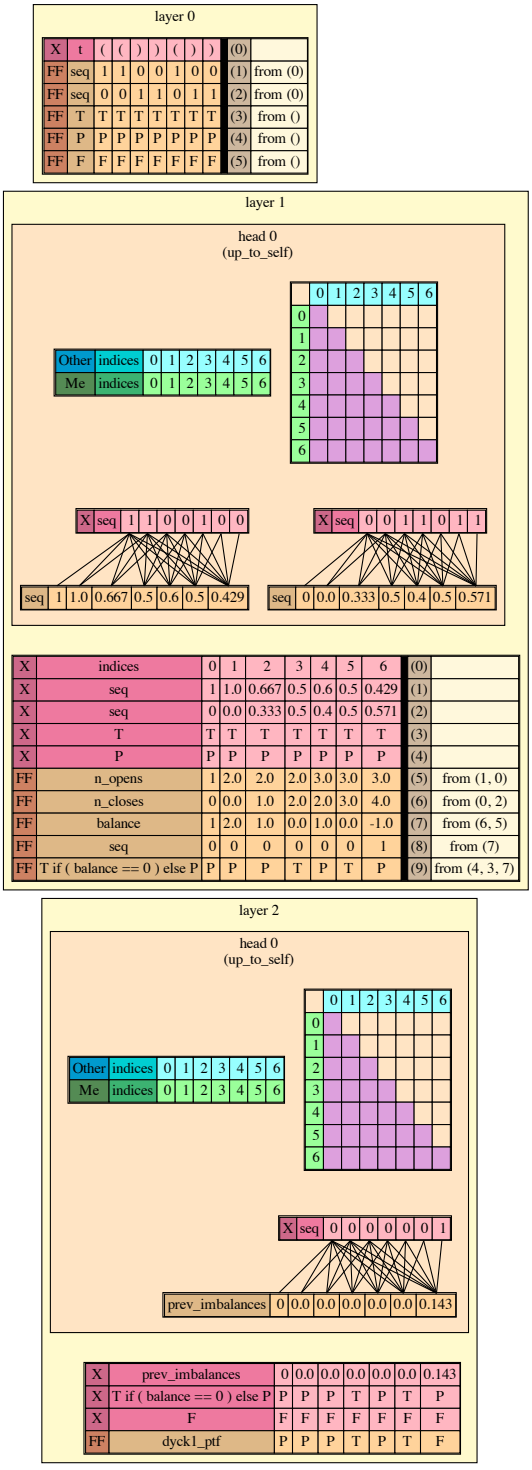

Figure 18: Computation flow in compiled architecture from RASP solution for Dyck-1, for solution shown in Figure 15. Applied to the unbalanced input sequence "(()())".

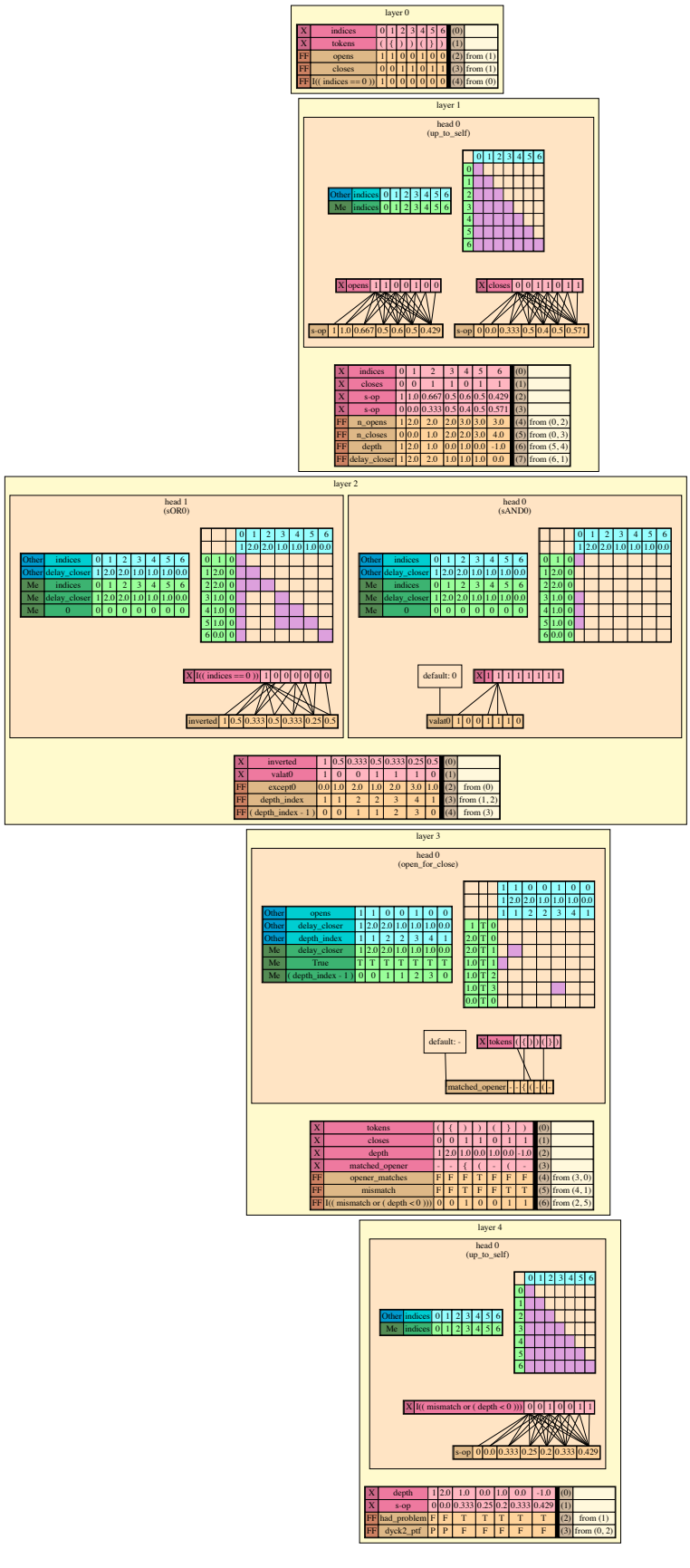

Figure 19: Computation flow in compiled architecture from RASP solution for Dyck-2, for solution shown in Figure 8. Applied to the unbalanced and 'incorrectly matched' (with respect to structure/pair-matches) sequence "())()".