RASP cheat sheet

RASP values are not mutable, but variable names can be reassigned. **The input sequence** refers to the value on which the s-ops and selectors are evaluated. It can be either a list of atoms (of the same type), or a string. When given as a string, it is treated as a sequence of strings of length 1.

Before s-ops and selectors

Atoms are ints, floats, strings, and booleans. They use similar manipulations to python (except exponentiation, which uses `a^b` instead of `pow(a,b)`). There are no bitwise manipulations. Examples: • `"a"+"b" = "ab"` • `3^2 = 9` • `"abc"[0] = "a"` • `True or False = True` •

Lists, Dictionaries exist purely for organisation.

- Creation: `l=[1,2]` and `d={1:2}`, or:
- `range(4)=[0,1,2,3]`.
- Indexing: `l[0]=1`, `d[3]=4`.
- Dictionary keys can be any atom.
- List elements and dictionary values can be any value.
- List concatenation: `[1]+[2,3]=[1,2,3]`.
- List zip: `zip([1,2],[3,4])=[[1,3],[2,4]]`.
- Dictionaries cannot be combined or zipped.

Dictionaries and lists can be used to make new dictionaries and lists with python-like list/dict comprehension.

- `[b+1 for b in [1,2,3]] = [2,3,4]`
- `[b+c for b,c in zip([1,2,3],[4,5,6])] = [5,7,9]`
- `{k:v+2 for k in {1:2,3:4}} = {1:4,3:6}`

S-Ops and Selectors

The built-in s-ops

- `tokens("hey") = [h,e,y]`
- `indices("hey") = [0,1,2]`
- `length("hey") = [3,3,3]`

Input types There also exist `tokens_str`, `tokens_int`, `tokens_float`, and `tokens_bool`, which assume different atom types in the input. Initially, `tokens = tokens_str`. You may want to change the REPL running example to the right type.
Elementwise operations: the same operations which combine atoms. Booleans can be converted to integers using `indicator`. There is also a ternary operator.

- `(indices*3)("hey") = [0,6,9]`
- `(tokens+"a")("hey") = [ha,ea,ya]`
- `(indicator(tokens=="e"))("hey") = [0,1,0]`
- `(tokens if indices%2 else "a")("hey") = [h,a,y]`

Select and Aggregate

`Select` and `aggregate` work together to make new s-ops.

Select You can think of the first and second s-ops given to `select` as the *input description* and *output description*, respectively. Selectors define which "input values" will be relevant in creating the new "output values" in `aggregate`. Examples:

- `select(indices,indices,<)("hey") =` $\begin{bmatrix} F & F & F \\ \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \end{bmatrix}$
- `select(tokens,tokens,==)("eek") =` $\begin{bmatrix} \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & F \\ F & F & \mathbf{T} \end{bmatrix}$

Aggregate averages the selected input values for each output position, as shown in Figure 1.

- Optional third parameter `v`, returned for each output position with no selected input positions. `v=0` if not given. E.g.: for `load5=select(indices,5,==)`, `aggregate(load5,tokens,"z")("hey")=[z, z, z]`.
- For output positions with exactly one selected input position, passes input value without attempting to average. This allows passing non-number values: Take `flip_s(indices,length-indices-1,==)`, for which: `flip_s("hey")=` $\begin{bmatrix} F & F & \mathbf{T} \\ F & \mathbf{T} & F \\ \mathbf{T} & F & F \end{bmatrix}$
  Then `aggregate(flip_s,tokens)("hey") = [y,e,h]`.

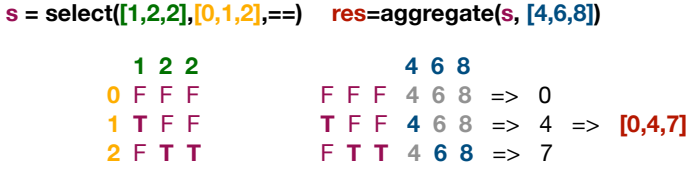

Figure 1: Underlying behaviour of `select` and `aggregate` on specific input sequences (not quite RASP: actual `select` and `aggregate` get s-ops and selectors). `select` marks for each **output position** all **input positions** satisfying the given comparison `==`. `aggregate` uses **s** as a filter over the **input values**, averaging only the selected **values** at each position in order to create its output, **res**. Where no **values** have been selected, aggregate substitutes 0 in its output.

Selector Width

How many input positions have been selected for each output position. For example:

- `selector_width(flip_s)("hey") = [1,1,1]`
- `selector_width(select(tokens,tokens,==))("eek") = [2,2,1]`

Other conveniences

For loops exist purely for code organisation. Can loop over dictionaries or lists only. Example syntax: `for i in [1,2] { a = a+i; }`.
Functions. Example syntax: `def foo(a) {return a+3;}`.
Loading Files store code in files with ending `.rasp` and use `load "[filename]";`, e.g.: `load "paper_examples"`. Variables beginning with _ are private to the file.

The REPL

Display and Commands

s-ops and selectors are displayed with a running example, showing the output of each s-op and selector. The example can be set for both together or independently:

- `set example ["hello","world"]`
- `set s-op example "hello"`
- `set selector example "world"`

It can also be toggled on or off, together or independently:

- `examples on`
- `selector examples off`

Additionally, the s-op example printouts can be made more verbose, showing aligned with input for clearer view:

- `full seq display on`
- `full seq display off`

You can exit the REPL using `exit`, `quit`, `exit()`, or `quit()`, or—in some systems—Ctrl+D.