1 Startup

Start. The generator takes a `,` separated list of actor files followed by a `,` separated list of def input files. They each are all lumped together.

The first actor's name, is the starting actor. The `go_act` function loop through all actors with this name.

All comand line arguments are store in the starting node instance as named entries. They are `${0}, ${1}` variables. To access these variables else where, prefix it with the starting actor's name like `${.main.1}`.

2 Variable

Variable names. The `${name}` gets replaced by the value of the variable `name`. To escape the `${`, use `$${` The case conversion letter `c` like `${name}c`, captilize the variable's value. The `${.arg}` is the value of the argument passed from the previous actor. The `${eval}$`, it replaces the eval with the `strs` function of the value of `eval`. That is to get a code block from a separate file and print it `C ${code_block}$`.

2.1 Purpose

The use cases.

Output print variable value.

Match compare value.

2.2 Special

Special variable names are prefixed by (.).

Global .Node.item.var The item index name of a Node.

Window .actor, the def of the actor.

Collections .Json, loaded json file.

Counters .+, loop counter.

Depth .depth, the actor stack depth.

Arg .arg, argument passed from previous actor.

Conditional .0, first or rest of loop counter.

Eval $, the content is re evaluated.

Optional ?, no error on var.

2.3 Errors

Variable name errors. The errors land up in the generated code to track down the error. Some commands make use of the `s_get_var, strs` functions that would return the error, but the commands ignore them. The errors are printed though.

3 Actor

The actors The actor are like functions that can be called and a case like statement that matches. The match is (var exp string), the string can have variables in it. Actors of the same name, are the case items. They are given an input node to operate on. The actor has a list of commands it runs through.

3.1 Purpose

Use cases.

Navigate call actor with a def.

Collect collect defs or strings.

Limit break out of loops.

Print print output text.

3.2 Name

Command names.

All actor call with all nodes of type.

Its actor call with defs related to current node.

Du actor call with the current node.

C print output line.

Cs print output with no new line.

Unique end actor if item is not unique.

Collect add def to a collection.

Group add strings to a group.

Break break out of the actor.

Include include file context to output.

Out delay or omitting output based on further output.

New add new node to input data.

Refs run loader refs after adding new nodes.

Var create new variable in the current node.

3.2.1 Var

Var command. This creates a named entry in the the current node's instance. The `Var foo = bar`, sets the variable. To access it, `${foo}` The `Var .list_act.foo = abar`, set the variable in the node instance that is current in the `list_act`. The current actors are on the stack. To access it, `${.list_act.foo}`, or when on that node instance (back to the list_act), `${foo}`.

Also has a `regex` that can break down the string as named entries.

3.2.2 Collect

Collection commands. These are to collect data for later use. They denormalizes the input to be more elegant for the generator. The `Collect` and `Hash`, store the current node's instance. The `Unique` and `Group` store strings - does not duplicate. The `Hash` can be accessed as a var `${.Hash.A.Open.foo}`, the others by the `All` command.

3.2.3 Break

Break command. The actor loop `go_act` works like a `case` switch. Actors of the same name, are the `when` cases. The command loop `go_cmds`, loops through the commands in that actor. The `All, Its` loop, calls the `go_act` function in a loop. The `Du` command call another actor with `go_act`. To break out of the `go_cmds`'s loop, it uses a break to end the loop. The `Break cmd` command and `Unique` does that. To break out of the actor loop, it returns 2. The `Break actor` does that. That ends up in the `go_act`, that end it. To break out of the `Its, All` loop, it returns 1. The `Break loop` command does that. The actor loop `go_act` would return 1 if its return is 1. The return 1 will end up in the calling `Its, All` command that will continue with the `go_cmds`. The generated code loop functions would return the result if `!= 0`. The other implementations make use of a depth value that gets inc/dec. It can go back further. The problem is the `Du` command `Do/Jump` that may or may not be between the loops.

3.2.4 Condition

Break condition. The `Break cmd on_error ${val}`, evals the variable string with the `strs` function. If the variables are missing, it would break out of the `go_cmds`. The `no_error`, would break when the variables are present - doing checks.

3.3 Call

Actor calls. The `All, Its and Du` commands, calls the `new_act` function to set up a new actor window on the stack. It passes the match data `(variable, eq, value)` and `arg` string. The value is evaluated from the current node instance. The `Du` command calls `go_act` with the current node instance, the others, the generated code that call `go_act`. The `go_act` function uses the new node instance. The match uses this instance and return if the match failed. Then it loops through all actors with its given name. Each of these actors, have there own match data and skips the ones that do not match.

3.3.1 Loop

Loop counter. The `All, Its` command calls `new_act` first that sets the next actor's counter to -1. The loop calls the `go_act` function, that increments the counter on match. The `${.-}` is the counter value and `${.+}`, the counter +1. Also `${.0.string}` for first (if counter is 0) and `${.1.string}` for rest. The value is `string` The `Du` inherits this value.

3.4 Match

Actor matching. Actor have a case like match on all the actors of the same name. `Actor list_act Node name = tb1`, here it matches the varable `name` to `tb1` The `&=` would be false if the previous one failed. The |`=` would be true if the previous one was true. The variable has a `?` option like `name? = tb1`. This would fail if `name` does not exist. No error is printed and the global errors flag is not updated - not seen as an error.

3.4.1 Matching

Match cases.

Equal =, var equal to value.

In in or has, var is in a list

4 Input

Input files.

4.1 File

Input files. The input files are word based separated by tabs or spaces. The last column can be a variable string `(V1)`, that is the string to the end of the line. There is one whitespace between the previous word and it. Use a padding word before it to get all the columns alligned if needed.

4.2 Other

Other input. The Json, Yaml and Xml are addons that operate the same way that the rest does. May need some more work here.

4.3 Errors

Load errors. The input file loader, prints errors as it goes along, mainly the parent and refs. The run time only checks these, but does not generate errors.

4.4 Types

Data type

Word C1, word.

String V1, string to end of line.

Ref R1, link to top level comp - Find - needs a Ref.

Local F1, link to local comp - same parent - needs a Ref.

Indirect L1, link to child of previous link - R1 for first, L1 for chain - needs a Ref2.

Multi M1, ref to local node of an element of R1 of a node that has a L1 ref - needs a Ref3.

Nest N1, control field of a nested node.

4.5 Nest

Node nesting. A control field of a nested node. The value 1 is for the top level, 2, next level down and so on. This is to create a tree from one node type. To navigate to the nodes one level down, use `Its group.right`. To navigate one level up, use `Its group.left`. The `Its group.up` goes to the node above it of the same level. The `Its group.down`, to the node below. The value 0 is for nodes that do not form part of this set. There can be more than one control field for different tree layouts.

```
----------------------------------------------------------------
Comp Frame parent Model FindIn
----------------------------------------------------------------

    Element group      N1 WORD       * search navigation group index tree
```

5 Window

Actor stack windows

5.1 Purpose

Use cases.

Store stores values needed.

Stack window are stored on the calling stack.

Access access to stack items.

5.2 Name

Window variables.

name actor name

cnt loop counter

dat node instance

attr node variable

eq equation

value compare value

arg argument passed from previos actor

flno line number of the calling actor

is_on out delay is on

is_trig out delay is triggered

is_prev previous actor has trigger

on_pos cmd index for trigger

cur_pos current cmd index

cur_act current actor index

6 Refs

refs The `R1` is a ref to another node, `F1`, ref to local node, `L1`, ref to local node of the `R1`, M1, ref to local node of an element of `R1` of a node that has a `L1` ref.

Sample of of a units file.

```
----------------------------------------------------------------
Comp Attr parent Type FindIn
----------------------------------------------------------------

    Element table    R1 Type             * Pointer to (Type).
    Element relation C1 WORD             * Relation type
    Element name     C1 NAME             * Colomn name.

Ref table Type check

----------------------------------------------------------------
Comp Where parent Type
----------------------------------------------------------------

    Element attr     F1 Attr      * Field name
    Element from_id  M1 Attr      * From id

Ref     attr Attr check
Ref3 from_id Attr attr Attr table check
```

The `Ref3` uses the `M1` field `from_id`. The link goes to node of type `Attr`. It uses the `attr` field to get to `Attr` node. In that node it uses the `table` field to be used as the parent `(Type)` to find the `Attr` in. The `from_id, attr` can be different node types. The refs run in a sequence at `Element` level. First it does the `F1, R1` ones, then the the `M1, L1` ones.

Sample of def file.

```
--------------------------------------------------------
Type User User
--------------------------------------------------------

Attr Contractor_employee     view contractor_name

Where contractor_name  id_number = contractor_id

-------------------------------------------------
Type Contractor_employee Contractor Employee
-------------------------------------------------

Attr . . id_number
```

If this data had to be loaded into a database, the foreign links need to be populated. Need select statements to get to the id's for this. The loader's refs does this.

```
Attr.tablep = Select id from type where name = 'Contractor_employee'
Where.attrp = Select id from attr where name = 'contractor_name' and attr.parentp
Where.from_idp = Select id from attr where name = 'id_number' and attr.parentp = A
```

To use the links from the `Where` node, use `attr,from_id,id`. To use it from the `Attr` node, use `Where_attr, Where_from_id, Where_id`. These are the reverse links that loops to match. The `Its` cmd will get them all, The variables will get the first one.

The `L1` is a simpler model of this, it uses the `R1` instead of the `F1` to get to the parent. The `F1` share the same parent. The `R1` finds the parent - top level nodes.

The `check` on the refs, means it is an error if it does find the link. A `(.)` here, means it is optional link. The value then also need to be a `(.)` if it is optional. If the value if different, then it is an error if not found. A `(?)` here means link if can, but no error if not.