KnitKit: A flexible system for machine knitting of customizable textiles - Supplemental Document

GEORGES NADER, Panasonic RnD Center Singapore, Singapore University of Technology and Design
YU HAN QUEK, Singapore University of Technology and Design
PEI ZHI CHIA, Singapore University of Technology and Design
OLIVER WEEGER, Technical University of Darmstadt, Singapore University of Technology and Design
SAI-KIT YEUNG, Hong Kong University of Science and Technology

ACM Reference Format:
Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, and Sai-Kit Yeung. 2021. KnitKit: A flexible system for machine knitting of customizable textiles - Supplemental Document. *ACM Trans. Graph.* 40, 4, Article 64 (August 2021), 9 pages. https://doi.org/10.1145/3450626.3459790

ABSTRACT

This document provides details about the implementation of the *KnitKit* system.

S.1 IMPLEMENTATION DETAILS

Our implementation of the *KnitKit* system consists of two independent modules: The *KnitNet* generator and the machine instruction generator.

The first module handles the geometry processing part of the *KnitKit*. It takes as input a triangular mesh (.obj or .ply), two texture images (.png), one for the stitch pattern and one for the yarn material, and an input vector field for the knitting direction (in .csv format). The input triangular mesh should contain the texture coordinates in order to allow the system to map the stitch pattern and yarn material to the geometry. In case an input vector field is not provided, the system can compute one by evaluating the gradient of low order eigenvectors of the mesh Laplacian (as stated in the paper). Once the input is specified, this module will generate the *KnitNet* according to Section 5 of the main manuscript. The output *KnitNet* is serialized in the EDN format [1] (Extensible Data Notation) and written to disk. This module is implemented in C++ and includes a GUI in order to visualize and inspect each stage of the *KnitNet* generation (Fig 1).

The second module that generates the machine instructions is implemented in the Clojure programming language. In our current implementation, users interact with it through a REPL process, executing functions which transform the data through the different intermediate stages. This interactive process allows for a tight feedback loop when selecting various library configurations and inspecting intermediate outcomes. Users may also choose to compose a fixed pipeline of functions and parameters into a static command line tool, which can be simply executed on the input file that contains the *KnitNet*. The module begins by parsing the EDN file into a native graph data

[1] https://github.com/edn-format/edn

Authors' addresses: Georges Nader, Panasonic RnD Center Singapore, , Singapore University of Technology and Design, georges_nader@sg.panasonic.com; Yu Han Quek, Singapore University of Technology and Design; Pei Zhi Chia, Singapore University of Technology and Design, peizhi_chia@sutd.edu.sg; Oliver Weeger, Technical University of Darmstadt, , Singapore University of Technology and Design, weeger@cps.tu-darmstadt.de; Sai-Kit Yeung, Hong Kong University of Science and Technology, saikit@ust.hk.

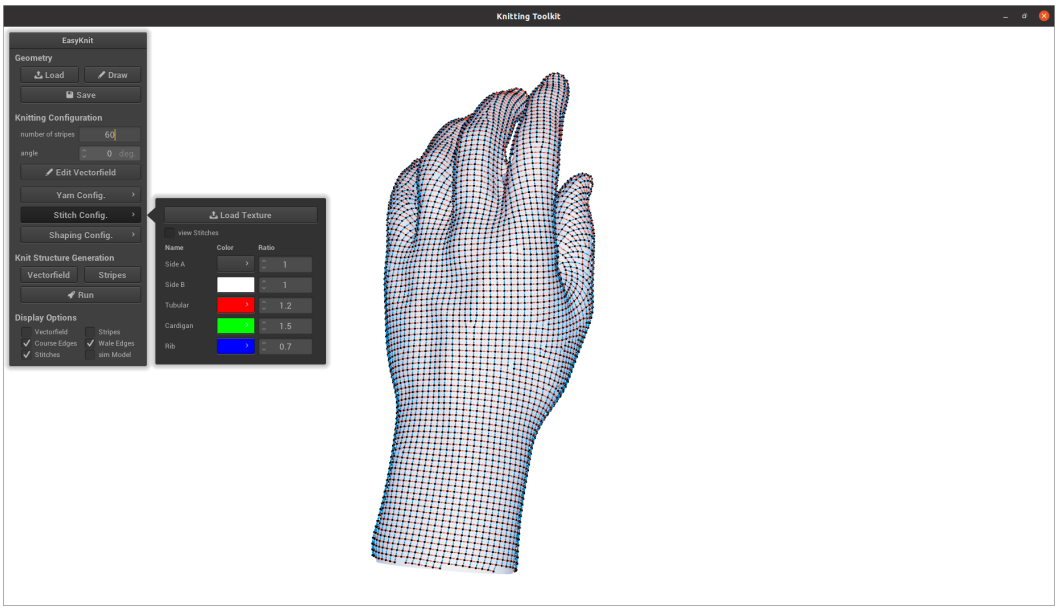

Fig. 1. The *KnitNet* generation module.

structure, converting it into the *action graph* format, and then translates the action graph into machine instructions according to a set of graph transformation rules and routines. This process is independent from the initial input and design stage, i.e., any other program that generates a suitable EDN file can interface with it equally well. In the next section, we will showcase some of the rules and routines that were used to produce the examples in the main paper. The entire set of rules and routines are included in the GitHub repository.

S.2 THE EDN FILE FORMAT

```
{:version "0.1.1"
 :knitnet/courses
 [{:c/id 0
   :c/material :DJ
   :c/cnx [[3 12 35] [:OUT 36 36]]
   :c/stitches [#:st{:id 1186 :loc 12 :type [1] :next 1275 :pos [0.737709 1.00564 0]}
                #:st{:id 1086 :loc 13 :type [1] :next 896 :pos [0.716478 1.00629 0]}
                #:st{:id 1190 :loc 14 :type [1] :next 792 :pos [0.695629 1.00617 0]}
                #:st{:id 1189 :loc 15 :type [1] :next 791 :pos [0.674832 1.00441 0]}
                ... ]}
  {:c/id 1
   :c/material :DJ
   :c/cnx [[0 12 35]]
   :c/stitches [#:st{:id 1587 :loc 12 :type [1] :next 1186 :pos [0.737829 1.02267 0]}
                #:st{:id 1087 :loc 13 :type [1] :next 1086 :pos [0.717335 1.02432 0]}
                #:st{:id 1088 :loc 14 :type [1] :next 1190 :pos [0.694781 1.02378 0]}
                #:st{:id 1187 :loc 15 :type [1] :next 1189 :pos [0.676237 1.02266 0]}
                ... ]}
  ... ]}
```

Fig. 2. Snippet of an EDN file containing a *KnitNet* (truncated).

Figure 2 is an excerpt from the EDN serialisation of a demo *KnitNet*. Its content reflects the *KnitNet* description presented in section 4 of the main paper. The file contains a sequence of courses under

the key `knitnet/courses`. Each course is assigned a unique `id` and contains a list of connections to other courses, as well as a sequence of stitches.

Lines 3 to 10 describes the course with `id=0`. It is connected to two other courses as specified in the `cnx` key at line 5. Stitches in the inclusive span 12 to 35 are connected to the course with `id=3` and the stitch at `loc=36` is connected to the special tag id `OUT`, indicating that it is a bind-off stitch. Line 6 to 10 contains the sequence of stitches of the course. Each stitch is assigned a unique `id` and a stitch `type`. The `st/next` key specifies the id of the wale-adjacent stitch in the connected course. Additional optional keys can also by included. For instance, here we include the `st/pos` key for visualisation purposes.

The combination of course material and stitch type defines its *template*. Here, the `DJ` value of the `c/material` key specifies a Double Jersey Jacquard stitch pattern, with the `st/type` key being interpreted as either of the two yarn colors in the jacquard pattern.

S.3 GRAPH TRANSFORMATION RULES

```
1  (ns knitkit.knet.rules
2    (:require [knitkit.knet.combinators :as cmb]
3              [knitkit.knet :refer [defrule]]))
4
5  (defrule LINEAR
6    "A simple linear reduction rule:
7    A -> B -> C -> D
8    a ------> x -> d"
9
10   ;; Template graph (Left)
11   [[A [B]]
12    [B [C]]
13    [C [D]]
14    [D]]
15
16   ;; Context graph (Middle)
17   [[A <a> a]
18    [C <c> x]
19    [D <d> d]]
20
21   ;; Replacement graph (Right)
22   [[a [x (cmb/chain-edges
23             ~[A B] ~[B C])]]
24    [x [d ~[C D]]]])
25
```

Fig. 3. Linear transformation rule.

As a first example, Figure 3 shows the linear transformation rule. The definition of a rule uses the macro `defrule` and requires three arguments representing a template graph, a replacement graph, and a context graph.

The template graph is defined between line 11 and 14. Line 11 denotes that node **A** is connected to another node **B**, with no special restrictions placed on their nodes or edges. Similarly, lines 12 to 14 indicate that node **B** is connected to node **C**, and that node **C** is connected to a final node **D**. The replacement graph is described between lines 22 and 24. Lines 22 and 24 indicate that node **a** of the replacement graph is connected to node **x** and that node **x** is connected to node **d**, respectively. In the replacement graph definition, the syntax `~[A B]` denotes a run-time substitution of the corresponding edge `A->B` in the template graph. The form `cmb/chain-edges ~[A B] ~[B C]` at line 22-23 denotes that upon matching a subgraph in some action graph G, the edges in G corresponding to **A->B** and **B->C** in the template are passed as arguments to the `chain-edges` function. This function chains their actions together and returns a new edge containing the chained actions. This newly created edge then connects the nodes of the transformed graph corresponding to **a** and **x** in the replacement graph.

The graph transformation algorithm that we use in our implementation requires an additional context graph to be specified. It describes the relationship between the template and replacement graphs. For instance, lines 17, 18 and 19 indicate that node **A** is transformed into node **a**, node **C** into node **x**, and node **D** into node **d**, respectively.

To summarize, the rule defined in Fig. 3 would substitute a subgraph of 4 nodes **A**, **B**, **C**, **D** with another of 3 nodes **a**, **x**, **d**. The information in nodes **A** and **D** will be copied to **a** and **d** respectively and the actions in the incoming edges of **B** and **C** will be chained together and stored in the incoming edge of **x**. Connections to other nodes outside the subgraph are handled by the algorithm according to the context graph.

```
29 (defrule NARROWING-R
30   "Delete a right-bindoff node X.
31
32   C <---- o ---> c
33   ↑              ↑
34   |  X           |
35   |  |           |
36   B <---- o ---> bx
37   ↑  |           ↑
38   | /            |
39   |/             |
40   A <---- o ---> a
41 "
42   [[A
43     [B {:edge/matcher cmb/reg-edge?}]
44     [X {:edge/matcher cmb/bindoff-edge?}]]
45    [B [C]]
46    [X] [C]]
47
48   [[A <a> a]
49    [B <b> bx]
50    [C <c> c]]
51
52   [[a [bx (cmb/splice-bindoff
53             ~[A B] ~[A X] :Rside)]]
54    [bx [c]]])
55
```

```
57 (defrule WIDENING-R
58   "Delete a right-caston node X.
59
60      C <--- o ---> x
61      ↑             ↑
62      |             |
63      B <--- o ---> bx
64     /↑             ↑
65    / |             |
66   |  X             |
67   |                |
68   A <------ o ---> a
69 "
70   [[A [B {:edge/matcher cmb/reg-edge?}]]
71    [X [B {:edge/matcher cmb/caston-edge?}]]
72    [B [C]]
73    [C]]
74
75   [[A <a> a]
76    [B <b> bx]
77    [C <c> c]]
78
79   [[a [bx (cmb/splice-caston
80             ~[A B] ~[X B] :Rside)]]
81    [bx [c]]])
82
83
```

Fig. 4. Left: narrowing rule, right: widening rule.

Figure 4 shows the definitions of a widening and a narrowing graph transformation rule. In the narrowing rule (left), lines 42 to 46 describe a template graph where a node **A** is connected to nodes **B** and **X**. The `edge/matcher` condition on the edges constrains the match to subgraphs such that the connection from **A** to **X** is a bind-off type edge (corresponding to **OUT** connections in the input KnitNet EDN.) Unless otherwise specified, all template edges created by `defrule` have a `reg-edge?` matcher. This rule aims at removing the node **X** and transforming the nodes **A**, **B** and **C** into **a**, **bx** and **c**, respectively. As indicated by the call to `splice-bindoff` in line 52, a `bindoff` action is added to the list of actions in the incoming edge of **bx**. The widening rule (right) works in a similar fashion. In this case, the rule aims to remove a `START` node **X**, and add `caston` actions to the list of actions in the incoming edge of **bx**.

Figure 5 shows the transformation rule that we use to create holes in the knitted object. The rule handles the transformation in three sections. First, it replaces the **L1**, **X**, **R1** nodes with a single **lr1** node (line 41-42), combining the actions in the incoming edges of **L1** and **R1** with a bind-off action specified by **X**, and storing the result in the incoming edge of **lr1**. The way those actions are combined is defined in the function `hole-edge-bottom` from lines 5 to 18. Secondly, it replaces the **L2**, **R2** with the node **lr2** and sequences the respective actions in the incoming edges from **L1** and **R1**. (lines 41-42). Finally, it replaces the node **B** and incoming edges from **L1**, **Y**, **R1** and **B** nodes with a node **b**, with a single incoming edge from **lr2** that contains the necessary actions (lines 45-46). Ultimately, this rule transforms a graph representing a topological hole to a linear chain of nodes.

```
(defn hole-edge-bottom
  "combine left, middle (bindoff) and right edges on the bottom of a hole."
  [le me re]
  {:pre [(= 1 (count (:edge/acts le))
             (count (:edge/acts me))
             (count (:edge/acts re)))]}
  (let [las (:edge/acts le)
        ras (:edge/acts re)
        mes (:edge/acts me)]
    (cmb/crg-case
      :> (cmb/assoc-acts le
           (cmb/chain-acts las (cmb/bindoff> mas ras)))
      :< (cmb/assoc-acts re
           (cmb/chain-acts ras (cmb/bindoff< las mas))))))

(defn hole-edge-middle ...)

(defn hole-edge-top ...)

(defrule KNIT-HOLE
  "A user defined rule"
  [[A [L1]
    [X {:edge/matcher bindoff-edge?}]
    [R1]]
   [L1 [L2]]
   [R1 [R2]]
   [L2 [B]]
   [Y [B {:edge/matcher caston-edge?}]]
   [R2 [B]]
   [B]]
  [[A  <a>  a]
   [L1 <l1> lr1]
   [R1 <r1> lr1]
   [L2 <l2> lr2]
   [B  <b>  b]
   [R2 <r2> lr2]]
  [[a   [lr1
         (hole-edge-bottom ~[A L1] ~[A X] ~[A R1])]]
   [lr1 [lr2
         (hole-edge-mid ~[L1 L2] ~[R1 R2])]]
   [lr2 [b
         (hole-edge-top ~[L2 B] ~[Y B] ~[R2 B])]]
   [b]])
```

Fig. 5. A hole rule.

S.4 ROUTINE LIBRARY

When the input action graph G is transformed to the canonical 2-node form $\overline{G}$, we extract the sequence of actions contained in its edge. The routine library is then used to generate the machine instructions from each action. In our current implementation, we make an abstraction over these operations, by having the routines generate a list of *moves* that the machine can execute and then converting those moves into the actual machine language. This allows us to extend support to other machines in the future, as it only requires implementing new conversions from the moves without having to redefine the routines.

We have implemented the following moves for the Shima Seiki MACH2XS machine:

(1) KNIT : perform knitting operations on needle bed
(2) MISS : move the yarn carrier without knitting
(3) TRANSFER : transfer stitches between needle beds
(4) YARN-OUT : Bring a yarn carrier out of operation
(5) YARN-IN : Bring a yarn carrier into a specified bed location

We then transform those moves to the Shima Seiki KnitPaint format using the functions in Figure 6.

```clojure
(ns knitkit.shima.converter
  (:require
   [knitkit.model :as model :refer [machine-code]]
   [knitkit.shima.structure :as ss]
   [knitkit.shima.config :as config :refer [apply-config]]
   [knitkit.shima.racking :as racking]))

(defmethod model/machine-code [:shima :MISS]
  [{:shima/keys [yarn-mapping] :as cfg}
   {:keys [:yarn/yarn :bed/span] :as move}]
  (let [rows [{:body (repeat (span/width span) 16)
               :bias (first span)
               :ops  {:R3 (get-in yarn-mapping [yarn])}}]]
    (apply-config cfg move rows)))

(defmethod machine-code [:shima :KNIT]
  [cfg move] ...)

(defmethod machine-code [:shima :TRANSFER]
  [cfg move] ...)

;; API
(defn into-structure
  [config moves]
  (->> moves
    ;; ==== Pre-process ====
    (derive-carriage-returns :>)
    (squash-yarn-outs)
    (combine-non-overlapping-moves)
    (ensure-final-carriage-position)
    ;; ==== Convert to structure rows =====
    (mapcat #(machine-code config %))
    ;; ==== Post-process ====
    (remove-empty-rows)
    (add-yarn-feeder-pts)
    (add-fixed-rib-and-bindoff)
    (ss/structure)))

```

Fig. 6. Implementations of moves in the KnitPaint format.

Below are examples of some routines that implement the `KNIT-COURSE` action.

Figure 7 shows the routine that generates the moves necessary to knit a course with a Knit-miss Jacquard stitch pattern. We used this action to knit the examples shown in Fig. 16 and 17 of the main paper. From line 14 to 16, the routine accesses information from the input machine state and action. First, it gets the existing islands (line 14), then calculates their span (line 15), i.e., their needle locations on the bed, and finally an array of boolean values that indicates which yarn to display on the outside of the fabric (line 16). In this case, the contents of this array are the values of the `type` key from the input EDN of Fig. 2. Then, it defines a series of moves (lines 17-25) by generating `KNIT` type moves for each yarn on the front and back beds. For each move, it generates the sequence of bed stitches by mapping the true and false values in the pattern array to either front knit, back knit, or miss (float) stitches. On line 19, the function `derive-kb` is threaded through the list of moves. It inserts any necessary kickback moves (of move type `MISS`) by looking at the state's yarn carrier poses, derives the knitting direction for each of the moves and associates a `crg/dir` key to them. The resulting modified pose is used to compute a new machine state (line 27). Finally, the function returns the new state and the modified moves.

Figure 8 shows the implementation of the routine for the CMYK Inlay stitch pattern, which is used to knit Fig. 14 in the main paper. This routine takes in a `channels` parameter, which is a map from yarn color (`Cyan`, `Magenta`, `Yellow`, `Black`) to a boolean pattern array. It takes the most

```
1  (ns knitkit.model.actions
2    "Utilities for defining actions + default routine definitions"
3    (:require
4     [knitkit.model :refer [defaction with]]
5     [knitkit.model.span :as span]))
6
7  (defroutine KnitMissJacquard
8    "Tubular knit miss jacquard template.
9    `pattern` is a seq of booleans, where
10     true  -> `outer-yarn` shown on outside of fabric
11     false -> `inner-yarn` shown on outside of fabric"
12   [{:keys [:bed/pose] :as state}
13    {:keys [pattern outer-yarn inner-yarn] :as action}]
14   (let [island  (calc-island action state)
15         span    (island-span island)
16         pattern (or pattern (span/cycle-in span [true false]))
17         moves   (with {:move/type :KNIT
18                        :bed/span  span
19                        :crg/speed (or (:crg/speed act) :Body)}
20                   (with {:yarn/yarn outer-yarn}
21                     {:bed/bed :F, :bed/stitches (map {true :kf false :miss} pattern)}
22                     {:bed/bed :B, :bed/stitches (map {true :kb false :miss} pattern)})
23                   (with {:yarn/yarn inner-yarn}
24                     {:bed/bed :F, :bed/stitches (map {true :miss false :kf} pattern)}
25                     {:bed/bed :B, :bed/stitches (map {true :miss false :kb} pattern)}))
26         [new-pose moves'] (threadcat derive-kb (:bed/pose state) moves)
27         new-state         (update state :bed/pose new-pose)]
28     [new-state moves']))
```

Fig. 7. Routine that implements the KNIT-COURSE action for the Knit-miss Jacquard stitch pattern.

prominent of the channels and uses it as the base double-layer jacquard pattern along with the `White` yarn. The rest of the channels are used to knit inlays on top of the base jacquard stitches (lines 37-44). Lines 47 to 80 are the definition of the moves, which are split into the jacquard and the inlay stages. The complexity of this routine shows how users can make experimentally-obtained tweaks to improve the stability of the resulting fabric. For example, the jacquard pattern is overwritten with a fixed sequence on the left and right edges (lines 50-51) to reduce stitch defects. To minimize the presence of long float yarns, we also insert tuck stitches at random locations in the inlay courses (lines 71-74). This routine also demonstrates the use of the key `:shima/ops` in the moves (lines 52 and 65), as well as directly specifying KnitPaint bytecode 117 in line 73 instead of a generic stitch alias. The ability to cross the abstraction boundary and specify these fall-through values makes it easier for the expert user to iterate on a routine definition, and can always be refactored to be machine-independent at later stages.

S.5 INSTRUCTION GENERATION EXAMPLE

Figure 9 shows a complete example for generating the machine instructions from an input *KnitNet*. First, the input *KnitNet* is loaded and converted into an action graph (line 14 and 17, respectively). On lines 21-24, we then apply the graph transformation rules to generate the sequence of actions. We then check whether the transformation algorithm has led to the canonical graph $\overline{G}$ (line 26). On line 28, we extract the sequence of actions from the edge of $\overline{G}$ and then generate the necessary low-level machine moves to produce those actions (lines 36-40). For that purpose, the initial state of the machine is set on line 31. Finally, the `into-structure` function on line 44 transforms the moves into the KnitPaint format supported by the Shima Seiki machine.

```
(defroutine CMYK-INLAY
  [{:keys [:bed/pose] :as state}
   {:inlay/keys [channels] :as action}]
  (let [span          (island-span (calc-island action state))
        [max-k max-v] (apply max-key
                             (fn [[k v]]
                               (reduce (fn [acc x] (if x (inc acc) acc)) 0 v))
                             channels)
        jq-ground :White
        jq-figure max-k
        jq  (get channels jq-figure)
        cmy (dissoc channels jq-figure)
        [new-pose moves]
        (threadcat derive-kb pose
          (with {}
            ;; ===== Jacquard base =====
            (let [jq (-> jq
                         (u/replace-start [true false true])
                         (u/replace-end [false true false]))]
              (with {:shima/ops base-ops
                     :move/type :KNIT
                     :crg/speed (or (:crg/speed act) :Body)
                     :bed/span  span}
                {:yarn/yarn    jq-figure
                 :bed/stitches (span/cycle-in span (map #(if % :kf :kb) jq))}
                {:yarn/yarn    jq-ground
                 :bed/stitches (span/cycle-in span (map #(if % :kb :kf) jq))}))
            ;; ==== Inlays ====
            (for [[i [in-yarn sseq]] (map-indexed vector cmy)
                  :when (some true? sseq)]
              (with {:move/type :KNIT
                     :bed/span  span
                     :shima/ops inlay-ops
                     :yarn/yarn in-yarn
                     :bed/bed   :F}
                (let [tuck-freq 9
                      rand-offset (rand-int tuck-freq)]
                  {:bed/stitches
                   (map (fn [m t]
                          (if (and *enable-tucks* (= m 117)
                                   (zero? (mod (+ rand-offset t (* 2 i)) tuck-freq)))
                            :tf, m))
                        (map #(if % 117 :miss)
                          (let [border (assoc [false false false] i true)]
                            (-> sseq
                                (u/replace-start border)
                                (u/replace-end border))))
                        (range))}))))))
        new-state (assoc state :bed/pose new-pose)]
    [new-state moves]))
```

Fig. 8. Routine that implements the KNIT-COURSE action for the CMYK Inlay stitch pattern.

```clojure
(ns knitkit.app.demo
  (:require
   [knitkit.parsers.course-graph :refer [parse-file]]
   [knitkit.knet :as knet        :refer [from-course-graph collapse]]
   [knitkit.knet.rules           :refer [default-ruleset]]
   [knitkit.model.actions :refer [compile-actions add-waste-and-drawthread add-porte-bindoff]]
   [knitkit.shima.converter         :refer [into-structure]]
   [knitkit.shima.config :as config :refer [with-yarn-mappings default-config]]
   [knitkit.shima.io     :as sio    :refer [write-dat! write-img!]]
   [knitkit.utils :as u]
   [knitkit.viz :as viz :refer [view!]]))


(def demo-course-graph
  (parse-file "resources/demo.edn" {}))

(def demo-knet
  (from-course-graph demo-course-graph))


(def demo-collapsed-knet
  (collapse {:rules    (default-ruleset)
             :strategy knet/clump-strategy}
    demo-knet))

(knet/collapsed? demo-collapsed-knet) ;; => true

(def demo-actions
  (knet/extract-actions demo-collapsed-knet))

(def machine-config
  (with-yarn-mappings
    (default-config)
    {:Body [:L 6 6]}))

(def demo-moves
  (compile-actions
    machine-config
    (->> demo-actions
      (add-waste-and-drawthread :FB)
      (add-porte-bindoff))))

(def demo-structure
  (into-structure
    machine-config
    demo-moves))

(comment
  (view! demo-structure)

  (let [filename (u/timestamped "demo")]
    (doto demo-structure
      (write-dat! (str filename ".dat"))
      (write-img! (str filename ".dat.png"))))
  ,)
```

Fig. 9. Instruction generation example.