Building on predefined real tuple operations and automatic functorial substitutions to simplify programming

Fritz Mayer-Lindenberg, TUHH

.. the methods used in the design of the 'π–Nets' language .. which was required to

- support <u>numerical</u> applications on digital computers (signal processing, robotics etc.),
  particularly high perf. embedded comp. apps on FPGA/SoC(FPGA+ARM) NWs
- support numbers represented by various codes including non-standard ones
- support compilation for simple FPGA based processors (small memories, no caches)
  support heterogeneous networks of FPGA based processors, distributed memory
  define and support particular heterogeneous networks of FPGA/SoC/processor nodes
- provide basic real time control for IO operations

Extra requirement: .. besides supporting readable and well-structured programs

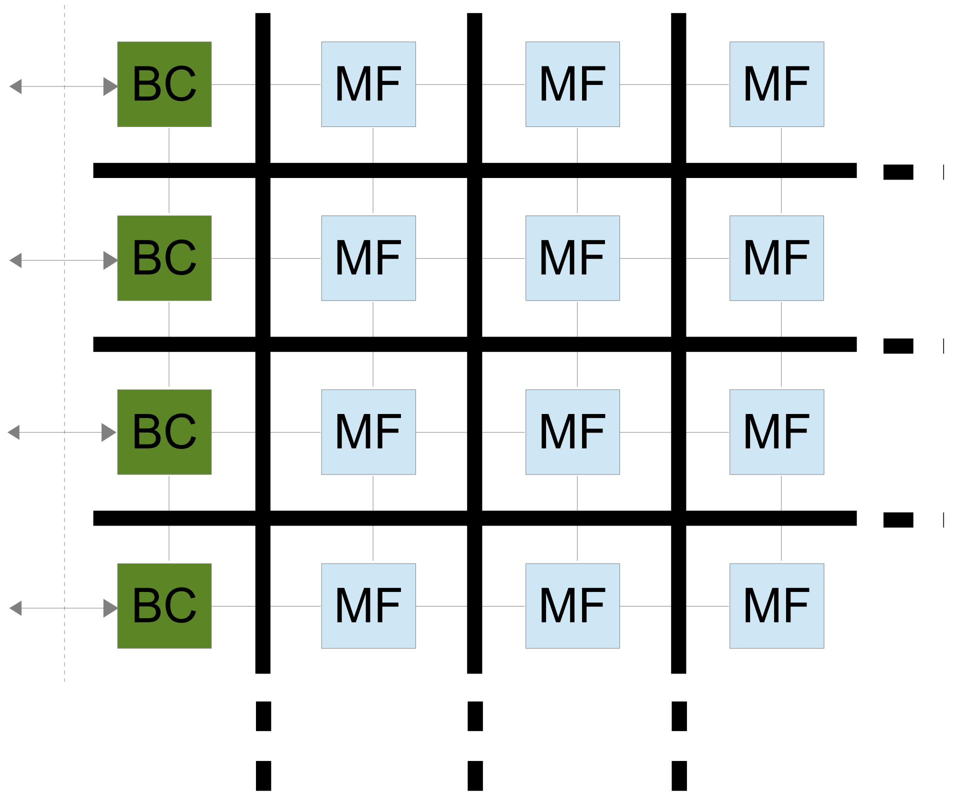
do so in a simple way to arrive at a small yet powerful programming tool !

'simple' here means .. the language being small and easy to learn (syntax&semantics)
with a few concepts, types and constructors only, avoiding choices
high abstraction, short (not cryptic) programs, *not: easy compilation*
.. easy transfer to the execution environment, debugging/simulation

....see SoC documentation for special motivation !! .. several std languages reclaim simplicity

Configurable 'logic': the FPGA

= single-chip integrated reservoir of many simple logic 'cells'



multi-function MF cells have indi-vidually configurable functions, no sequential control

border BC cells are configurable interfaces to the external pins

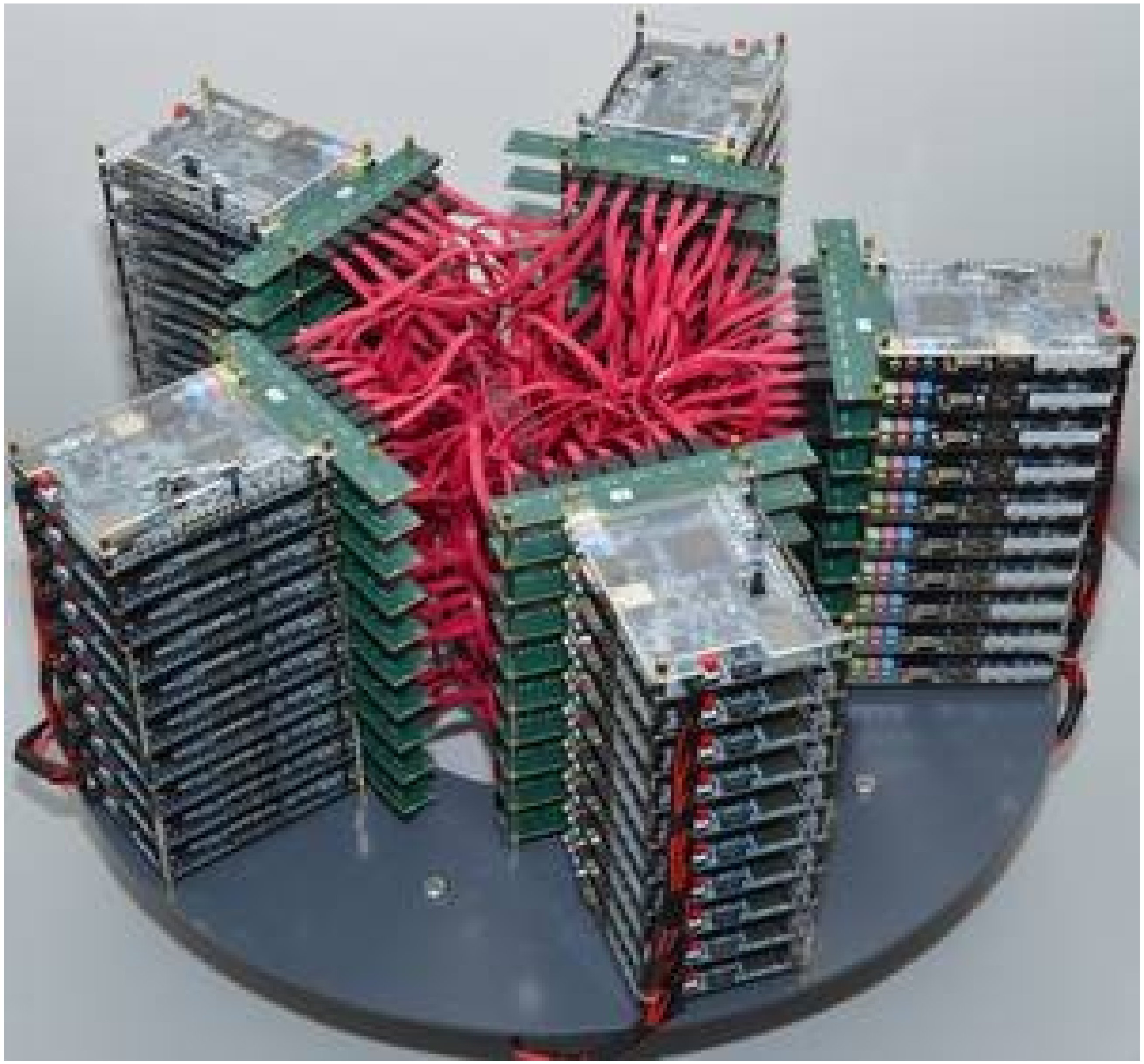
electronically configurable wiring network,
separate clock signal networks

at lower density: complex arith-metic functions, memory blocks, fast serial interfaces

The FPGA cells can be connected via the wiring network into compute circuits (ALUs) for arbitrary number codes and be equipped with control circuits to become programmable processors.

Example: 100 k MF cells
220 multipliers
250 2kB RAM blocks
8 serial IFs, DRAMC

A possible target : the ER4 experimental computer at the TUHH



- a general purpose parallel computer for research/education easy partitioning, IO extensions, rewiring

- 50 SoC nodes each containing

    2 ARM cores
    FPGA
    2GB DRAM
    16GB flash

- host PC or terminals connect to any node

- configurable to hold 800 soft processors for various codes plus100 ARM cores

! A single SoC node
Is a non-trivial target.

The Fifth programming language (1984-1997) .. proving simplicity is possible

- supporting several microcontrollers/-processors and networks of such through preconfigured versions of the compiler, integrated assemblers

  6801/5/11, 8031, 6809, Z80, 6502/816, 8086, 68k, C166, 78310, Transputer, 56k, 2181, C40, Sharc, 34010, PPC

- machine oriented providing bit field operations, easily ported due to HW abstraction by programming for a stack oriented virtual machine

  functions associated to executing processors, comm. by remote function calls

  infix option for numerical expressions, number code input (fixed/float)
  automatic conversions for local and remote function calls

  Types: bit fields (fixed size, bool, byte), integers (same/double code size, idx), floating point numbers (1 fmt/p, sgl/dbl)
  .. strings = char array literals, aut.outp. .. no automatic error handling .. int/fp automatically tracked, fp lits.

- multiple threads, sync, process variables (besides global i/f vars/arrays, locals)

- small self-contained integrated programming, compiler and debugging environment
  implemented in Fifth and compiled to native code for the PC … easy to compile
  individual native code generation for all processors of a Fifth target

  interactive testing by linking the execution environment to the PC

-!! successfully used in a number of industry projects .. individual licences, not commercialized

.. and in research: automatic parallelization, multivalued ftcd, exotic procs and FPGA

.. port to Win95++ difficult (integration with C code) , pNets implemented in C, ext.ed.

Programs formally define composite computations ('algorithms'):

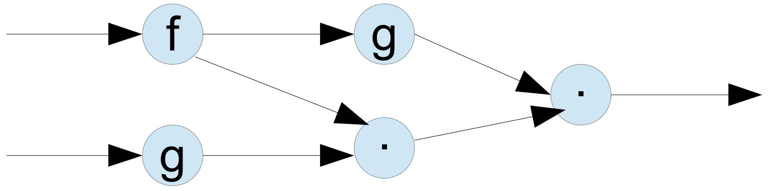
- Algorithms are given by a composition scheme (a directed graph S) with a set C of compute nodes and an assignment $\eta: C \to O$ of nodes to types of computational operations with compatible signatures. The composition scheme involves

  compositions of functions and operations ($f \circ g$)
  branches to select between (the results of) alternative sub computations
  can be recursive
  no schedule, no storage involved, complexity = $\#C_{eff}$ .. can be data/control dependent

  Algorithms not performing memory access operations but only numeric operations define/compute pure functions mapping numbers to numbers.

  .. are supposed to do so after a finite number of selected steps ('FNA').

  Operations to be composed are the operations specified for the available data types.



- FNAs will generally suffer from

  errors due to applying operations/functions to data outside their domains

  approximation errors due to the finiteness requirement

  interpolation errors once functions are represented by finite tables only

  rounding/overflow errors depending on the number codes used on a machine

Could some subset of the mathematical notation in textbooks be part of a p.language?

.. for numerical computations

* Some nice notational features:

- values defined by expressions don't need a termination symbol
- names for functions/data can freely use special symbols, Greek characters
- can use subscripts and superscripts, can chain comparisons
- formulas can extend above and below a baseline (integrals, fractions) .. hard to edit
- conditional, even recursive assignments extending over several lines .. hard to edit

* Mixing with natural language w/o extra indications

- formulas often embedded into nat. lang. sentences
- math. texts are edited for the human reader, headers, comments
- textual references to previous definitions/contexts

There are notions lacking a mathematical counterpart

- IO, communications, reading and writing variables, processes/threads, real time

.. and mathematical constructors w/o a counterpart on a computer

- defining infinite sets such as cosets of a vector sub space or infinite sub group
- defining sets of equivalence classes .. e. g. of algorithms
- using images of some infinite set under a mapping
- using existence results proved non-constructively
- defining a structure up to isomorphy only through a universal property .. e.g. $V \otimes W$

Remark on algorithms and proofs:

- algorithms can be extracted from constructive proofs which yet focus on verification
- mathematical reasoning tends to avoid numerical coordinates as far as possible

.. implement math/geom types

Methods to achieve simplicity

- restrict range of applications and targets
  choice of paradigms to identify/restrict programming task
- abstraction from HW&implementation (codes, serializing with memory)
- supply powerful predefined operations for the predefined data sets
  - predefine vector/tuple sets and tuple operations '+','-' .. overload them
- small number of predefined data types only, no(?) data set constructors
  - distinguish data types on predefined sets by their operations only
- small number of structures for program control, short readable syntax
- **apply functorial methods as high-level compile-time functions**
  - overloading symbols (to have fewer names of operations/functions)
- support required extras like parallelism, real time, target definition in
  the simplest ways, using hierarchy to cope with complex applications

π–Nets uses a unique type of real* number, operates on finite function tables.

*** .. extended by 'NaN' elements**

All data sets are predefined: **.. no Booleans** **.. no string operations**

fixed size tuples of real numbers (finite fixed size tuple valued function tables)

!! tuple/table entries may be 'invalid'

.. no 'row' or 'column' vectors no other predefined or derived data sets

predefined tuple operations: apply tuple as a function, composition .. (idx maps)

linear/multilinear vector operations

(using ideal real arithmetics) polynomial operations

interpolation operations

.. important basic selection .. set operations .. DSP, modeling, robotics

(extend by libraries of algorithms) relations .. used to test&branch

Algorithms are statically defined, pure program functions:

map tuples to tuples .. compatible with tuples applied as functions

may have invalid results .. can be tested for to branch

! always terminate .. to be vf'd by the programmer

.. maybe, with an invalid result .. recursion supported to limited depth only

As a simple substitute for data type definitions, program functions can be grouped into 'optype' definitions and then redefine/overload predefined op symbols such as '+' and '*'. The optype names are used to select the definitions to be applied.

* could use rational/algebraic/computable numbers instead, encode two numbers (scaling, error)

Tuple access operations: x an n-tuple of real numbers, m|n, m'=n/m

-	x	call the full tuple as an operand
-	x°y	composition of tuples, y entries are indexes < n (or tuples of such) most general
-	x.i	access i-th tuple component (a number), i.e. apply x as a scalar fct. n:x.i , x:1.i
-	x:m.i	apply x as m-tuple valued function, 'x.i' if ':m' is set as default m': index tpl m':x.i m':x:1.i
-	x.(i,j,k)	multidimensional indexing, x:m.(i,j,k) same as an m-tuple val'd fct. (s,t):(i,j) (r,s,t):x.(i,j,k)
-	x°y.i	index applies to y .. y could be a table of permutations
-	x.(i,j,k)°y	transposing indexes to x x°°y.(i,j,k)

.. distinguished as non-comp 'addressing' ops

Some arithmetic tuple operations: .. to be overloaded according to the x/y sizes

-	x y	apply x as a linear operator x @ y x ∧ y
-	x pol y	apply as a polynomial
-	x ipl y	interpolate from x as a function table

Further ingredients of πN-programs: 'otypes','atypes' (w.memory), application processes

- from the predefined operations, pure functions on numbers including constants can be statically defined through algorithms (notation uses blocks, lambda var's).

- a selection of pure functions can become an 'optype'. The names of the functions can then redefine (overload) previous definitions. Optypes can inherit from others.

- types of automata ('atype' definitions) are defined similarly but allowing for a state memory and read/write operations. 'atypes' can have sub automata and inherit.

- applications are defined as sets of communicating individual automata equipped with individual or predefined active control. Processes can be/have variables and sub automata, perform input and output with other processes. Process control can break up into several control threads that can be given individual timings. Communications withing the same process are implicit.

- the application processes are associated to number codes and executing processors and are compiled to individual control codes for the processors. Memory allocation follows the code distribution. Only the functions called in a process compile for the executing processor. The compiled code includes FPGA configuration, the distribution to soft processors and state initializations.

and: the available hardware for an application is structurally defined within a program (in an include file) as a hierarchic net list.

The notions of functors and of functorial substitution

In category theory, *functors* map objects of one category to objects of another, e.g. a vector space V to its dual V*, and morphisms between objects to morphisms between the corresponding objects in a way compatible with composition, e.g. a linear map $h:V\rightarrow W$ between vector spaces to the transposed map $h^T:W^*\rightarrow V^*$.

An algorithm specifying multiple compositions (a finite number) is then mapped to a similar algorithm in the target category, the 'algorithm' $f\circ g$ e.g. to the algorithm $g^T\circ f^T$.

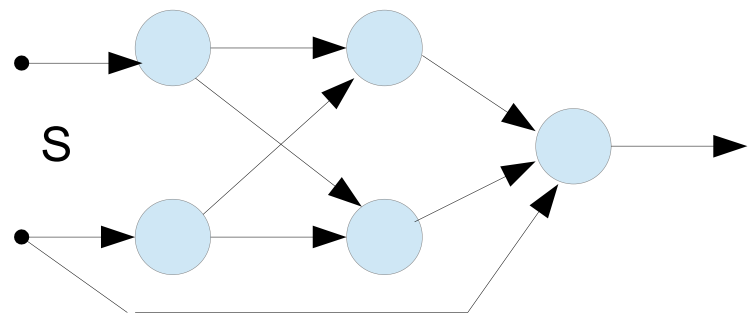
In this presentation, *functorial substitution* refers to a method applying to algorithms (defined in a programming language), namely the formal substitution of their operations by associated ones operating on other data yet maintaining the composition scheme. The substitution also applies to operations/functions with multiple arguments and to conditions, branches and recursions in order to cover all algorithmic structures.

Functorial substitution can also be applied if it is defined for the function computed by an algorithm, too (not only predefined ones) and is not a functor. Then for the function F defined by an algorithm A, the substituted algorithm A' need not be an algorithm for the substitute F'. If F is 'called' in another algorithm, there is the choice to either substitute F by F' or to substitute it by the function computed by A'.

The substitution may be applied to predefined functions by choosing algorithms for them. It may be an automatic compiler operation or be on explicit commands yet w/o needing to explicitly redefine the composition scheme, thus simplifying programming.

Functorial substitution

S: a composition scheme (a dir. acyclic graph w.IO)
C: set C of compute tokens in S

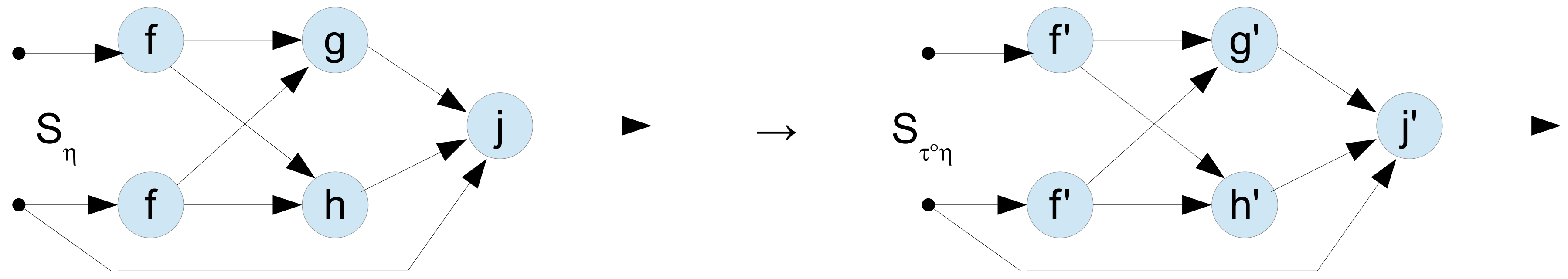


Every mapping $\eta: C \to O$
of C to a set O of compatible types of op's/rel's determines an algorithm S_η for a function F_η.

For an assignment $\tau: O \to O'$
of compatible op types *functorial substitution* with τ is the transformation of algorithms

$$S_\eta \to S_{\tau\circ\eta} .$$

Generally $F_\eta \notin O$ or $S_{\tau\circ\eta}$ is not an algorithm for τF_η.
Then substitute by $F_{\tau\circ\eta}$ instead of τF_η. .. except for a functor



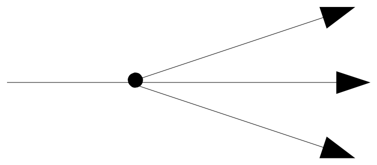
$f'=\tau f$ etc.

.. algorithm computing F_η .. algorithm computing $F_{\tau\circ\eta}$ $(\neq \tau F_\eta)$

Graphs and algorithms: .. non-std extensions to graphs, also useful for HW

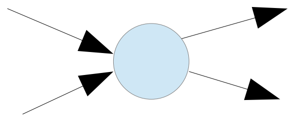
- composition schemes have open directed edges (graphs do not)

  can distribute node outputs to several inputs (fan-out)

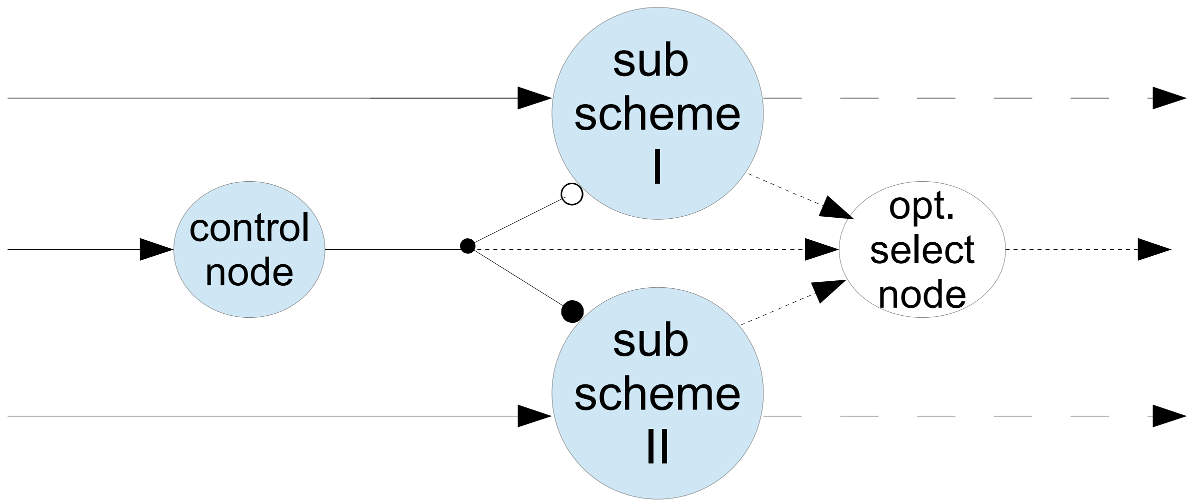


  are composed of elementary (single node) ones

  can be composed, even recursive, be hierarchic



- algorithms branch to alternative sub schemes according to relations evaluated at special control nodes; necessary to break recursions



Data dependent control selects a unique directed execution path through the sub schemes. All internal inputs to them are from nodes back on this path excluding sub schemes.

0. Mapping operations and functions to multiple data

Map operations/functions $f: P \to Q$ to $f': P^n \to Q^n$, $(p_0,..,p_{n-1}) \to (f(p_0),..,f(p_{n-1}))$

.. for $P = A \times B$, f has two arguments, and f' is defined to be a function on $A^n \times B^n$.
or more $\approx (A \times B)^n$

Example: the scalar add operation $+: \mathrm{IR} \times \mathrm{IR} \to \mathrm{IR}$ extends to the vector add operation.
... mpy ... * … to multiplying vectors by components

by composition, extend linear maps to tensors

LISP: use MAPCAR for this extension

π–Nets: automatic extension (overloads operations/functions to more tuple operations)

.. more options: $f(a, b) = (f(a,b_0), \dots , f(a,b_{n-1}))$ for $a \in A$ and $b \in B^n$ etc.

This substitution is (an extension of) a true functor. Substituting all operations in an algorithm A without branches for a function f by the corresponding tuple operations yields an algorithm A' for the function f' on tuples.

Equivalently, the individual $f(p_i)$ can be computed separately which also allows for a control flow with individual branch selections, i.e. by horizontally expanding the original algorithm instead of substituting the operations/functions therein.

1. Using functorial substitution to support various number codes

Numbers need to be encoded by bit strings before they can be digitally computed with.

enc: $\mathbb{R} \to B^*$	encoding function	partially defined
dec: $B^* \to \mathbb{R}$	decoding function	partially defined such that

rnd = dec°enc: $\mathbb{R} \to \mathbb{R}$ the 'rounding' function fulfils enc(r) = enc(rnd(r))
hence rnd°rnd = rnd

Rem.: rnd is a unique operation assoc.to enc/dec same domain as for enc
rnd/enc often derive from dec

Operations op: $\mathbb{R} \to \mathbb{R}$ are substituted by op'=enc°op°dec: $B^* \to B^*$ on the machine

r = rnd(r) $\Rightarrow$ dec(op'(enc(r))) = rnd(op(r)) (substitution inserts rounding)

Operations op: $\mathbb{R}\times\mathbb{R} \to \mathbb{R}$ etc. are handled similarly, op'=enc°op°(dec$\times$dec): B*□B*→B*.

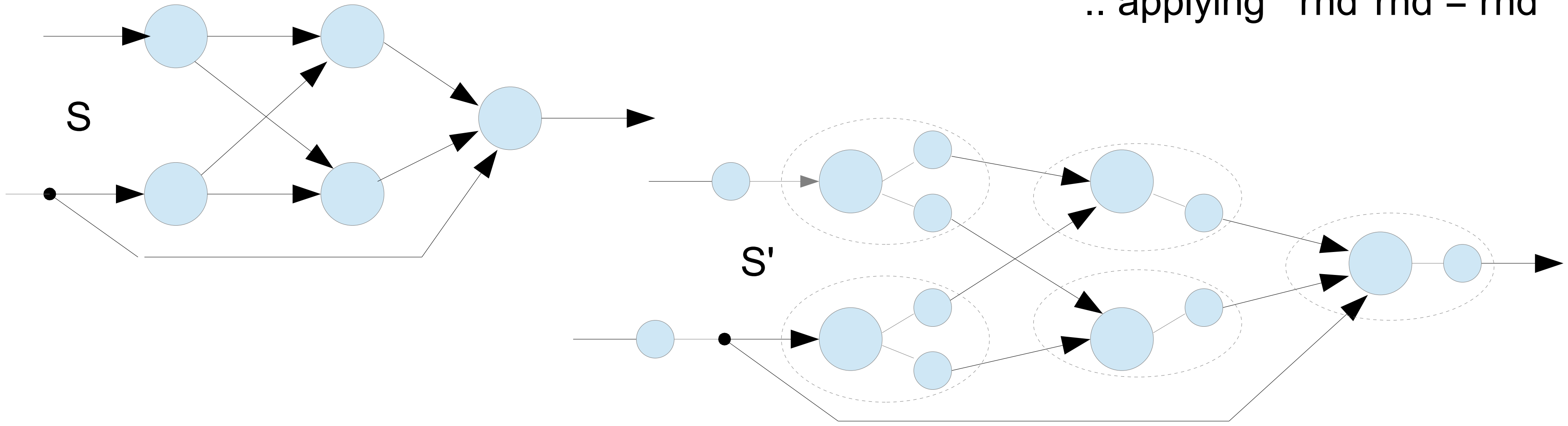
Conditions c $\subset$ $\mathbb{R}$□$\mathbb{R}$ are substituted by c' = {(p,q)|(dec(p),dec(q))$\in$ c} $\subset$ B*□B*.

Numeric algorithms (compositions of real operations) are executed on the machine by performing functorial substitution of every operation op on the reals by the corresponding op' on number codes rounding its output. The same substitution also applies to the composed real function. In general, substituting within the algorithm does not yield an algorithm for the substitute of the composed function without intermediate rounding (the 'fused' one).

If a program function g is called within an algorithm for f, g is not substituted as a fused composite operation with added rounding, but by the function computed by the substituted algorithm for g similarly to expanding the algorithm for g into the algorithm for f, flattening the hierarchic definition. Machines may be built to compute certain fused functions, however.

'rnd' insertion as a graph transformation: - can study functotial subst. of ext'd graphs

- input roundings cancel except for ext.inp.
.. applying   rnd°rnd = rnd



'rnd' insertion for conditions (example uses πN syntax):

… { **if** f(x) < 1   **then**   g(x) **else** h(x) } …

- glues partial functions g,h depending on data (not necessarily 'continously')

- insertion of rnd's changes the condition to   'rnd f'(x) < 1'
  .. can result in 'discontinous' behavior not close to the original
  .. if 'f(x) < 1' is evaluated at compile time to always hold, the branch disappears

Standard PLs distinguish different types of numbers by the codes to be applied (**int**,**float**).
.. no abstract reals, rounded substitutes for the real operations only
.. mathematical algorithms substituted by hand (conscientiously ..?)

π-Nets: supports several standard and non-standard encodings including

I32, X16, X35, V144, F32, F64, G45 .. to be expanded

by their (unique) rounding operations only and as attributes to the abstract computations performed by its processes telling the compiler how to substitute operations. The rounding operations are available as predefined real operations allowing to test for the applicability of the code and to model the operation of the executing machine. For every encoding, errors can and need to be estimated and tracked through an algorithm, maybe using simulation.

The I32 code applies to integers only and does not perform rounding at all. An extra integer divide operation is available to explicitly 'round' real numbers to nearby integers.

By default, number codes extend to tuples of real numbers using tuples of codes of their entries (substitution **0.**). n-Tuples can be applied as functions to integer indexes in the range 0...n-1 (yield invalid data otherwise). As primary data tuples can use codes of their own, e.g. an n-tuple of fixed point mantissas plus a extra exponent. The elementary functions typically add extra approximation errors. Certain composite operations can be implemented as 'fused' operations w/o intermediate roundings, e.g. the complex operations for V144 including the butterfly and the quaternionic ones.

Implementation in the π-Nets compiler:

- intermediate code (CDGF) DFTs represent real number operations
  may use associativity/commutativity for optimization etc.

  tuple operations not expanded into scalar operations, appear as tuple DFTs

  no expansion of function calls in the intermediate code

  number code selection is a CFDG attribute

- automatic insertion of rounding operations during interpretation of the intermediate code for the purpose of simulation or execution on the PC

  interpretation applied to constant folding and to external processes uses a high precision code type performing rounded operations from which the less precise application codes are rounded ($rnd_{app} = rnd_{app} \circ rnd_{max}$)

  automatic substitution of encoded operations during code generation and

  automatic insertion of code conversions for data transfers between processes

- storage for intermediate data and the variables of application processes:
  storage of numbers is in full internal precision for interpretation/simulation, but is for the required code word size only on the processors executing native code

  Rem.: external memory is a set of special function nodes storing/xferring msgs

Error handling and 'invalid' codes.

The automatic insertion of rounding operations during simulation or code generation may cause runtime errors at places that are not explicit in the program, such that the errors are not handled by the program. The same kind of error also occurs for the implicit code conversions when data are sent from some process to another one using different codes. In these cases the failing rounding operation (indicating that the number to be encoded is outside the encoded domain) returns an 'invalid' but does not break execution. Embedded applications often tolerate erroneous data.

All scalar operations on numbers are defined to yield an invalid result on invalid input. In a tuple, individual entries may be valid or invalid, and tuple operations defined in terms of scalar operations can still deliver tuples with some valid components. In order to maintain this functionality, number codes must provide at least one bit pattern for invalid data. The standard floating point formats provide NaN codes that can be used for this, and standard floating point units deliver NaN outputs on NaN inputs. If more codes are available for invalid data, they can be used to transport more information on the error.

As π–Nets only provides numeric operations and numeric or invalid output there seems to be no way to symbolically annotate its numeric outputs on some display. A simple trick is played to overcome this limitation. Strings like “abc” are allowed as literals for invalid data and sent to the predefined process writing to the display. This process interprets the invalid data as the command to print out the string or the error it represents. No special test for data to be valid is needed.

2. Substituting *predefined* operations by algorithms (composite operations)

Vector operations (e.g., linear mappings) and polynomial operations (evaluation, product, division) are defined in terms of the '+' and '*' operations of some base field/ring. Their substitution can be defined by a change of these ring operations.

From a given type of real numbers, complex operations can be defined on pairs of real numbers in the usual way, using the definition (a,b)*(c,d)=(ac-bd,ad+bc) as an algorithm. Complex n-vectors can be represented as size 2*n tuples, and the operation of multiplying a complex scalar to a complex n-vector becomes an operation multiplying a pair to a 2n-tuple with a 2n-tuple result. The real pair-to-vector multiplication can be overloaded with this complex operation without affecting the real scalar to 2n-vector operation.

This can be used to expand existing <u>predefined</u> linear and polynomial operations to other base rings while maintaining their usual names and calling syntax. The substitution is thus applied to a selected algorithm for the predefined operation but does not depend on it as long as no unfused coding is involved. The substitution of the operations within a previously defined program function is rarely needed, however, and does not need to be supported at all.

πN: Complex and quarternion operations are easily defined and packed into corresponding optypes; the same applies to other real algebras and even to redefining the real '+' and '*' by scalar modulo operations. Algorithms Selecting an optype overloading the '+' and '*' operations then dispose of the corresponding vector and polynomial operations as well.

Whether a 2n-tuple 'is' a real or a complex vector depends on the operations (optypes) applied to it only.

1) For 2n-tuples x,y their dot product 'x y' is the real number

$$\sum_i x_i \cdot y_i \quad .$$

If 'cpx ·' is defined and 'cpx' selected then 'x y' denotes the complex dot product. x,y become accessed as tuples of pairs x:2, y:2 .

2) 2n-tuples are applied as polynomials to a number x writing 'x pol y' to get

$$\sum_i x_i \cdot y^i \ .$$

After selecting 'cpx' 'x pol' applies to pairs z, using pairs (x_{2i},x_{2i+1}) as complex coeffs.

3) n^k-tuples x are applied to k-tuples of numbers as polynomials in k variables to get

$$\sum_{(i,j,k..)} x_{(i,j,k,..)} \cdot y_0^{\,i} \cdot y_1^{\,j} \cdot y_2^{\,k} \ldots \quad .$$

Selecting '+' and '·' operations mod(2) and n=2 2^k-tuples can be applied as Boolean functions to k-tuples with 0/1 entries.

.. many other examples: convolution algebras, Clifford algebras, finite fields

3. Automatic differentiation

$f: U \subset V \to W$ differentiable, with $V=\mathbb{R}^n$, $W=\mathbb{R}^m$ (real vector spaces)

$Df: U \to L(V,W)$, $Df(u)$ derivative of f in u (the approximating linear map) .. no coords.

.. D is not a functor

Sum:	$D(f+g)(u)=Df(u) + Dg(u)$	for $g: V\to W$
Product:	$D(h(f,g))(u)=h(Df,g)(u)+h(f,Dg)(u)$	for $g: V\to Y$, $h:W\times Y\to Z$ bilinear
Chain rule:	$D(g\circ f)(u)=Dg(f(u))\circ Df(u)$	for $g:W\to Z$
	$D(h(f+g))(u)=Dh(f(u),g(u))\circ(Df(u)+Dg(u))$	for $g: V\to W$, general h

→ derivatives of compositions can be computed from the $(f(u),Df(u))$ of the components
chain rule involves composition of linear maps (matrix product)
→ can substitute calls to functions f by calls to the (f,Df) evaluated according to the rules
at individual $u\in U$ to compute composition of function tables with functions/operations

Applications only requiring particular partial derivatives $Df(u)\cdot v$ don't need matrix products:

Define $TU=U\times V$ (the tangential bundle over U, i.e. the union of the $T_uU=\{u\}\times V$) and

$Tf: TU\to TW$ $(u,v) \to (f(u),Df(u)\cdot v)$

The composition rules simplify; the chain rule e.g. becomes $T(g\circ f) = Tg\circ Tf$.

→ can substitute calls to functions f by calls to the Tf evaluated with the simplified rules
(need to know the derivatives of the used elementary functions .. their approximations)

.. derive Hamiltonian vector fields from function

Functions $f:U\to W$ extend to functions with values in 'jet' spaces $U\times L(V,W)\times L_s(V^2,W)$, and automatic differentiation extends to higher derivatives:

e.g. $T(Tf)(u,v,x,y) = (f(u), Df(u)v, Df(u)x, Df(u)y+D^2f(u)(v,y))$.

Simplified higher order approximations through higher order derivatives:

For $U \subset V$, $TU = U \times V$ is the set of first order approximations $\gamma(0)+t\cdot\gamma'(0)$ to curves $t \to \gamma(t)$ at 0.

For $f: U \to W$, Tf maps the approximation to γ to the approximation of $f \circ \gamma$ at 0. $T(f \circ g) = Tf \circ Tg$.

Define $T_{(r)}U = U \times V \times \ldots \times V$ (r+1 factors). Then $TU = T_{(1)}U$. $T_{(r)}U$ is the fiber sum r times TU.

The $(u, v_1, ..., v_r) \in T_{(r)}U$ parametrize the rth order approximations $t \to u + t\cdot v_1 + t^2/2\cdot v_2 + ... + t^r/r!\cdot v_r$ to curves γ at 0, i.e. $v_i = \gamma^{(i)}(0)$.

Define $T_{(r)}f: T_{(r)}U \to T_{(r)}W$ to map the rth order (Taylor) approximation to γ to that of $f \circ \gamma$ at 0.

Lemma: For fixed u, $(T_{(r)}f)_i$ is polynomial in the v_j and the derivatives of $D^j f$ for $j \leq i$.
$T_{(r)}$ is functorial, $T_{(r)}(f \circ g) = T_{(r)}f \circ T_{(r)}g$.

Proof: $(u, \gamma^{(1)}(0), \gamma^{(2)}(0), ...) = T_{(r)}\gamma(0,1,0,0,...)$, need to compute $T_{(r)}(f \circ \gamma)(0,1,0,0,...)$, the $(f \circ \gamma)^{(i)}(0)$.

$(f \circ \gamma)^{(1)}(0) = Df(u)v_1$ with $u = \gamma(0)$, $v_1 = \gamma^{(1)}(0)$, $(f \circ \gamma)^{(2)}(0) = D^2 f(u)(v_1, v_1) + Df(u)v_2$,

$(f \circ \gamma)^{(3)}(0) = D^3 f(u)(v_1, v_1, v_1) + 3D^2 f(u)(v_1, v_2) + Df(u)v_3$ etc. by the chain rule … q.e.d.

Rem.: - for a diffeomorphism f $T_{(r)}f$ is a diffeomorphism, yet nonlinear on the fibers for r>1
- the functoriality of $T_{(r)}f$ allows for an automatic rth order differentiation $\to D^r f(v_1, .., v_1)$
- can be applied to numerically 'solving' first order ODEs.

Integration into a programming language:

- use a class of extended tuples and operations (a data type of tuple codes) define functions with automatic differentiation as functions of this type

- integrate A.D. into the language by just providing a calling syntax and implementing it into the compiler.

  This can build on a type of tuples of real numbers.
  It can extend to function tables and to all variants of A.D.
  The language is mostly untouched but gains functionality by compiler upgrades.

πN: use the name T'f for an algorithm f or a function table f

call to f or to T'f as needed

extensions to higher order approximations read T''f, T'''f … (not yet impl.)

Alternatively, the use of T' in the definition of f could be dropped to further simplify the notation. Its use enhances the expressiveness and somewhat simplifies compilation by indicating that there will be the need for extra data and allowing for a warning if a variant of A.D. is not applicable or not supported. The substitution of operations can be deferred to interpretation or code generation s.t. even the intermediate code compiled for an application can remain unchanged.

For a number tuple/function table f with k indexes, T'f will 'be' a tuple of k+1-tuples.
For an algorithm f with k inputs and m outputs, T'f has 2k inputs and 2m outputs.

Some applications: .. candidates for HL compile fcts

- Hamiltonian vector fields (classical mechanics, robotics)

  $H: V \times V^* \to IR$ $\quad X_H(q,p) = D_1H(q,p) - D_0H(q,p) \in T(V \times V^*)_{q,p} = V \oplus V^*$

  $q' = \partial H/\partial p(q,p) \quad p' = -\partial H/\partial q(q,p)$

- approximating integral curves c for 1st order differential equations

  $X: W \to W$ $\quad c: [a,b] \to W \quad c' = X°c$ $\quad$ unique for given $c_0 = c(0)$

  linear approximation for step size h: $\quad c_{i+1} = c_i + h \cdot c_i'$ $\quad$ with $c_i' = X(c_i)$

  same higher order using $c'' = (DX°c) \cdot c'$: $\quad c_{i+1} = c_i + h \cdot c_i' + h^2/2 \cdot c_i''$ $\quad$ with $c_i'' = DX(c_i) \cdot c_i'$

  etc. ('vertical' recursion) .. can use coarser steps

- backpropagation (m-layer perceptron networks, 'learning' weight matrixes A_i)

  $Y = T_m A_m \dots T_2 A_2 T_1 A_1 X$ $\quad X \in V$ input, A_i linear, $Y \in W$ output, T_i thresholds (non-linear)

  perform gradient descent on the A_i for $e_k = ||Y_k - S_k||^2$ for given series (X_k) and (S_k)

  .. apply A.D. to the T_i

4. Deriving error functions

Rounding errors reflect the approximate representation of input data and the results of operations. They are usually estimated by an upper bound but actually depend on the arguments of the rounded operation. For a program function, errors accumulate depending on the data (also via the branches taken that may depend on rounding). During simulation operations may be substituted by double operations one executed with the prescribed rounding and the other at the highest available precision.

A similar approach can be taken to analyze the approximation errors by simulation. They occur when the true result is the limit of a sequence of numbers computed by a recursive program function stopped after a finite number of steps. Ideally, the recursion delivers nested intervals around the limit. Achieving a desired precision will also depend on the precision of limits on the way and on the rounding errors after substitution.

If a finite computation aims at approximating a function f on an infinite domain, e.g. an open subset U of some real vector space V, U has to be replaced by a finite subset H. Then, approximate values of f on H are computed as inputs to an interpolation operation on these to get the approximation to f on U.

More generally, there is a finite-dimensional parameter space W, a linear 'interpolation'

$d: W \to F(U)$ and a (generalized) sampling operation $c: F(U) \to W$

such that $c°d = id_W$. This is similar to the coding of numbers. The 'rounding' $e = d°c$ can serve to substitute operations p of F(U) (like differentiation) by operations $p'=c°p°d$ on W.

Examples:

0) W is the space of functions on H, $F(U) \rightarrow W$ is restriction. The interpolated functions for a subspace $P \subset F(U)$ are in bijection to W under restriction.

1) U is the unit cycle in the complex plane and H is the set of n-th roots of unity. Interpolate with the (sin x)/x kernel. P is the space of complex polynomials of degrees < n and in bijection with the space W of their coefficients (DFT).

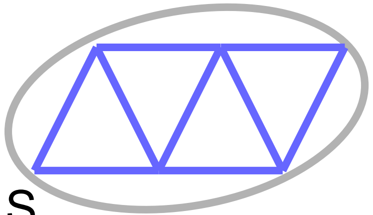
2) U is the sphere in IR^3, W the space of homogeneous polynomial functions of degree n on IR^3. Restriction to U is injective.

3) $F_r = F(U, \Lambda^r V^*)$ r-forms on U, $d: F_r \rightarrow F_{r+1}$ exterior differentiation

$\int_M : F_r \rightarrow IR,\ \theta \rightarrow \int_M \theta$ integration over r-dim. $M \subset U$

Stoke's formula: $\int_M d\theta = \int_{\partial M} \theta$ r+1-dim. M with boundary ∂M

Discretization: describe $M_0 \subset U$ as a simplicial complex, $M_0 = U_i S_i$



'encode' θ by the tuple of the $\int_S \theta$ over all r-subsimplexes S of the S_i

→ can compute integrals of θ over arbitrary sub unions of M_0 exactly from its 'code'

→ get encoded d operator by using Stokes formula to get the code for d (all $\int_{Si} d\theta$)

can use Whitney forms for interpolation

Remarks on the d-operator on forms and its discretization:

- The d-operator on forms does not depend on a choice of coordinates; the same holds for differential operators derived from d. The discretization transforms such operators to linear ones on tuples

- a prominent operator of this kind is the Laplacian on k-forms,

  $$\Delta\theta = {*d*d}\,\theta + d{*d*}\,\theta$$

  that also involves the Hodge-Operator $*$: $\Lambda^r V^* \rightarrow \Lambda^{n-r} V^*$ (n=dim V) defined in terms of a dot product on V. '*' needs to be 'encoded', too. The corresponding 'code' tuple spaces W_r and W_{n-r} are not isomorphic, however, as they should be. Instead, a dual cell complex and a corresponding d operator need to be used. The n-r-cells in the dual complex correspond 1-1 to the r+1 simplexes to which they are orthogonal w.r.t. the Euklidean dot product.

- The Maxwell equations describing electrodynamics can be expressed with the d operator. E,H are the electric and magnetic fields on IR^3 respectively, D,B,J the dielectric and magnetic flows and current density, and Q the charge density. E,H are time dependent 1-forms on IR^3, D,B,J 2-forms, and $D={*E}$, $B={*H}$ *up to constants*. Then (1) $dE = -\partial B/\partial t$, $dB=0$ and (2) $dH = \partial D/\partial t+J$, $dD=Q$. On the 4D spacetime with the extra coordinate t, define the Faraday tensor by $F=B+E\wedge dt$. Using the Lorentz metric to define '*', (1) and (2) transform into

  $$(1)\quad dF = 0 \qquad \text{and} \qquad (2)\quad d{*F} = Q-J\wedge dt \qquad \text{u.t.c.}$$

  Computing the forms can use discrete exterior calculus. This application and more suggest its support by special operations (Bossavit, Leok/Stern, Arnold).

An example: Musical Processes

```
.. uses predefined type mdo of MIDI automata with
.. associated constants    t1, t4, t8, t38, t16, t316     (duration codes),
..                         c 60, e 64, g 67, c' 72        (tone codes)
..                         band, pia, wood, x0            (instrument codes)

fct 1 1 fib            .. defines algorithm for Fibonacci sequence mod(24)
{    → i
     { → aa,bb,ii                              ,, end recursive Λ expression
          if   ii=0  then   aa
     else if   ii=1  then   bb
     else                   bb, aa+bb %24, ii-1 ←} (0,1,i)
}

apc mus  on host,                      .. main application process group
          (mdo) vc1   band, 55         .. midi sub automata
          (mdo) vc2     pia, 77, x0           .. receiving initial input
          (mdo) vc3   wood, 88                .. assoc. threads redef. 'apply'
          (mtm)   mt  0.125            .. mus. time increment
{
  #vc1   t1    24: {→i   fib(2i)+66, t16    fib(2i+1)+66, t316 }

  #vc2 14:{→i  fib(3i)+54, fib(3i+1)+58, t38    fib(3i+2)+60, t8 }   c,e,g,c',t2

  #vc3   7: {   c',t316   c',t316   c',t4   c',t316   c',t316  }         drum 100 c',t38
}
```