The Memtran language

– an optimized implementation of no-garbage semantics

Martin Nilsson

mani@memtran.org

GITHUB ADDRESS HERE

About me

- I'm a hobbyist, creating this programming language as a free software project – not an academic
- Currently not tied to any organization
- Probably spent far too much time on it...
- I hope you will be interested!

Memtran

- Compiled language (LLVM backend)
- Memory safe
- Imperative -- but certainly allows some aspects of OOP / FP
- Memory managed

  - No ”tracing garbage collection”

  - No ref count annotations on memory objects (but some flags on their "head")

  - ***"Custom memory management scheme"*** (More details later in presentation)
- Semantics:

  - "Assign-by-a-conceptually-unique-instance-of-the-whole-datastructure"

  - "Pass-by-a-conceptually-unique-instance-of-the-whole-datastructure” **OR** pass-by-reference
- Open source compiler (v0.1)

Benchmarks

Tour of the language

Dynamic size arrays

```
// Direct initialization -- nothing can be NULL
array1 ' arr f64 = [1.2, 3.4, 5.6, 7.8]

// Bypassing the initialization requirement
mu array2 ' arr int = [trash 100]
// Here we proceed with actual initialization
for it over array2 :                   // special for loop construct
    it = floor(random() * 100.0)
;

// "Assign-by-a-conceptually-unique-instance-of-the-whole-datastructure"
mu array3 ' arr f64 = array1
// Increasing the size dynamically
add_last(ref array3, 9.0)
```

Structs

```
type monster = monster{                  // Structs have a ”tag”
    id ' int
    hp ' f32
}

mu monster_array : arr monster = [
    monster{id = 1  hp = 100.0}, // "Struct expressions"
    monster{id = 0  hp = 50.0},
    monster{id = 2  hp = 80.0}
]

monster_array[1].hp = monster_array[2].hp - 5.0

/* Again, "assign-by-a-conceptually-unique-instance-of-the-whole-
      datastructure” */
mu another_monster_array ' arr monster = monster_array
another_monster_array[2].hp += 2.0

print(another_monster_array[2].hp) // will print: 82.0000
print(monster_array[2].hp)         // will print: 80.0000
```

”Variant-box types” (a.k.a. tagged unions)

```
/* Variant-boxes are (semantically) "containers holding one value,
    which value's type can be either of several types".
  -- so in Memtran, T / U is a completely different type from T or U... */
type my_variant_box_type = monster / f64 / arr int / nil

/* ”Up-conversion” is automatic (maybe not for v0.1 though) */
mu my_variant_box ' my_variant_box_type = 3.14

storetype my_variant_box case monster :
    /* 'my_variant_box' can be treated as downcasted/unboxed here,
            also on left hand side */
    my_variant_box.hp += 1.0
    print(my_variant_box.hp)
    println()
; case f64 :
    print_utf8("It's an F64!\n")
; default :
    print_utf8("Default case!\n")
;
```

”Pass-by-a-conceptually-unique-instance-of-the-whole-datastructure”

```
fn create_new_monster_from_old(mu m ' monster) ' monster :
    m.hp -= 50.0
    return m
;

new_monster ' monster =
    create_new_monster_from_old(monster_array[0])

print(new_monster.hp)          // will print: 50.0000

print(monster_array[0].hp)     // will print: 100.000
```

Runtime polymorphism

```
// Like an "abstract base class":
type animal = horse / lion / pig

/* 'default' EXPANDS to missing cases;
 no need to "maintain typeswitches" */
fn talk(a ' animal) :
    storetype a default :
        talk(a)  // overloaded function
    ;
;

type horse = horse{}

fn talk(h ' horse) :
    print_utf8("Neigh!\n")
;

type lion = lion{}

fn talk(l ' lion) :
    print_utf8("Roar!\n")
;
```

```
type pig = pig{number_of_ees ' int}

fn talk(p ' pig) :
    for i ' int = 1 to p.number_of_ees [
        print_utf8("Ee ")
    ; println()
;

fn random_animal() ' animal :
    switch floor(random() * 3.0) case 0 :
        return horse{}
    ; case 1 :
        return lion{}
    ; default :
        return pig{
            number_of_ees =
                floor(random() * 15.0) + 1
        }
    ;
;
animals ' arr animal =
    [random_animal() repeat 10]

for it in animals :  talk(it)   ;
```

Generics

```
fn(t) insertion_sort(
    ref array ' arr t,
    greater_than ' fn<t, t to bool> // First-class function
) :
    for i ' int = 1 to len(array) - 1 :
        mu j ' int = i
        loop :
                                                /* "infix sugar" */
            if !((j > 0) && (array[j - 1] `greater_than` array[j])) :
                break
            ;
            swap(ref array[j], ref array[j - 1])
            j -= 1
        ;
    ;
;

inline fn(t) swap(ref a ' t, ref b ' t) :
    temp ' t = a    a = b    b = temp
;

fn my_greater_than(a ' monster, b ' monster) ' bool :  return a.id > b.id  ;

insertion_sort(ref monster_array, my_greater_than)
```

More...

```
// Separate compilation module system
import "utf8.mh"

// Parametrized types
type my_array(t) = arr t

// Multiple return values
a ' i8, success ' bool = to_i8(31234789)

// Named arguments can optionally be used
concat(ref array1 = monster_array, array2 = monster_array)
```

Absolutely minimal initial overview on how to use Memtran

- **Destruction of data** ”held” by variables/parameters, no matter whether it is a ”single value” or a whole datastructure, is performed **automatically** at the end of the var's or param's scope.

  (In actual reality, things like *transfer* or *sharing* may make such destruction *unnecessary*, but then things are as they should anyway, in a way.)

  So you don't need to cater for that particular aspect of memory saving.
- Temporary data generated by expressions is also automatically destructed in the general case – i.e. as soon as it is not needed for computation anymore.
- You have to use the **mu** marker if you plan to mutate the data ”held” by a variable or parameter.
- When you want to mutate, through a *parameter*, the data ”held” by an *”argument slot”*, and *in-place*, you have to use the **ref** marker.
- And if you want to acquire some understanding about the actual **time costs** of operations, you can keep following these slides, to learn more about what the implementation actually does.

Why Memtran doesn't need a tracing garbage collector

- Due to the **special semantics** of this language, it is always possible to tell at **compile time** where destruction operations should happen in object code.

- It is easiest to understand why this is so when considering the ”Simplified Semantic Model” (see next slide).
  - The actual runtime characteristics are then probably easiest comprehended as a series of optimizations upon an imagined, maximally naive implementation of the Simplified Semantics.

Semantics vs. Implementation (1a)

Simplified Model Semantics – model of WHAT a Memtran program does:

(here in terms of some kind of idealized everyday physics)

- Memory conceptualized/imagined as consisting of a **”stack of elastic pockets”**. Each pocket holds exactly one instance of a ***value***, and each pocket is its value instance's sole owner.
- In a modelled program execution, whenever a variable or parameter is instantiated, a new pocket, with a newly created, assigned, value instance, is **pushed** onto this (conceptual) stack of elastic pockets, and the var/param instance is equivocated to it. (And conversely, going out of scope means **pop** pocket stack.)

  (Pocket stack pop = destruction.)
- Some value types range over values of **differing size** (however you define ”size”). Pockets designated for values of such types, can hold value instances of differing size, in a **balloon-like manner.**
- All expressions are (imagined to be) function calls returning new, unique, value instances. (Function's params and local vars are of course stack popped at return.)
- When a **mutation operation** changes the content of a pocket, the value instance previously held is automatically destructed, making room, conceptually, for the new value instance that is assigned. (Mutating subparts of pocket contents simply replaces a subpart instead.)

Semantics vs. Implementation (2)

Recap

- Trying to estimate actual **time cost** of operations (with regard to the actually existing implementation) from how the described ”simplified model semantics” could be implemented on a computer most simply, can be highly **misleading**.
- Time characteristics of the actual implementation should hopefully be better than what such a simplistic implementation would deliver.

Semantics vs. Implementation (3)

Implementation – HOW Memtran does what it does

- Values implemented as a ***"head only"*** are stored on the **execution stack**, and/or in **registers**.
  - For such "head only" values, **no structure sharing** is performed in the object code -- they are **always copied** when assigned or passed (non-ref) around.
- But values of certain other types are implemented as a ***head + heap memory pointed to*** (possibly *n* levels deep)**.**
  - Using **pragmatic compile-time heuristics**, the compiler implements assignment, and passing (non-ref), of such mostly heap allocated values, in terms of either
    - ***deep copying*** *(with associated novel heap allocation),*
    - ***borrowing***,
    - ***transfer***, *or*
    - ***sharing***.
  - Automatically generated, necessary **destruction operations** take this into consideration, while at the same time respecting the semantic model.
  - Due to **optimistic** application of the **borrowing** strategy, a so called **"ghost system"** is also implemented, which provides **fallback** in certain cases.

Summary of what the implementation actually does -- for performance-concerned programmers

- **A.** Values of **types that are or contain array types:**
  - **Var/param names** used as expressions, and which return such values, are simply considered **sharers that can transfer**. And so are all **indexing expressions** (i.e. array indexings, struct indexings, and ortype downcasts – or combos of those) that return such values.
  - **Passing to an immutable, non-construand, non-ref parameter** (of a global function) is **always** implemented in terms of **borrowing**. (But when the provider of the value is a global variable, ghost system may need to kick in later.)
  - **Immutable variables share** a provided value if an immutable var/param (or indexing thereof) is the provider.
  - In all other cases,
    - **transfer** is used when possible (but the compiler does not attempt this for global variables).
    - Otherwise, assignment or passing (non-ref) is performed as a **deep copy**.

B. Other values:

- All assigning and (non-ref) passing is implemented as **copying**.

Alternative summary

A. Values that are implemented as a **head only**

(numbers/bools, structs of numbers/bools, variant-boxes of numbers/bools – or structs or variant-boxes of those)

	Pass to immutable, non-construand, non-ref param (of global function)	Pass to other non-ref param	Assignment	Pass to ref param
Semantics	”By-whole-value” (like complete copying)	”By-whole-value” (like complete copying)	”By-whole-value” (like complete copying)	By reference
Implementation	Copy	Copy	Copy	Pointer (sometimes a ”mutlock” is set, too)

B. Values that are implemented as a **head + things pointed to**

(i.e. arrays, or structs or variant-boxes holding arrays)

	Pass to immutable, non-construand, non-ref param (of global function)	Pass to other non-ref param	Assignment	Pass to ref param
Semantics	”By-whole-value” (like complete copying)	”By-whole-value” (like complete copying)	”By-whole-value” (like complete copying)	By reference
Implementation	Head copy (sometimes some flags are set, too)	Head copy **OR** deep copy (see big table)	Head copy **OR** deep copy (see big table)	Pointer (sometimes a ”mutlock” is set, too)

Important inspirations (1)

(Turbo) Pascal

- The first PL I came in contact with
- Gear ratio between amount of learning needed, & creative/productive power, was high to begin with

  (Confession/clarification: for a long time, I simply had my arrays global and static, and didn't use pointers... so I never had any segmentation faults back then...)

Important inspirations (2)

Euphoria

- Somewhat popular language in the 90's
- Seen as "toy language" by some
- Initially an **interpreted** language
- Later an optimizing **compiler-to-C** was released
- Memory safe / memory managed
- Semantics: Assign-by-a-conceptually-unique-instance-of-the-whole-datastructure / pass-by-a-conceptually-unique-instance-of-the-whole-datastructure
- Elegant and very simple basic type system:

  integers, floats,

  sequences (basically: dynamic arrays with index type ”Any”)
- Sequences implemented as reference counted, possibly structure shared memory objects

  - Untested hypothesis: cache issues would make this comparatively slow on modern computers

Design guidelines

- What would I like to program in?
  - - Something as **simple** as the inspirations
  - - But without some of the limitations
- Can it be somewhat **fast** too?
- Well..

Improving on Euphoria (1)

- ***Assign/pass-by-a-conceptually-unique-instance-of-the-whole-datastructure***

  - was **PERFECT** to a large extent

    - No aliasing bugs

    - Encouraging functional thinking

    - But can I have **pass-by-reference as an alternative?**

Improving on Euphoria (2)

- Its ***type system*** was quite **PERFECT** in its own way

  - Sequences could describe the structurality of most kinds of data

  - Constants holding indices into a sequence used for struct-/record-like functionality

  - *"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."* -- Alan Perlis

  - Runtime checked "type functions" were also added to the language at some point

- Go for something more performant? A more static type system?

  - ***numeric types***

  - ***dynamic arrays*** with indices of specific specified type

  - ***struct types***

  - no "Object" type a.k.a. "any type"

  - but ***variant-box types*** providing almost comparable functionality

  - but we have to add **generics** then...

The "different" memory handling idea

- Basic design of the new language was more or less finished
- No implementation was available to benchmark

Arm-chair thinking about memory management

- Alternative I. Reference count annotations

  - Probably quite slow

  - Ref counts "on the heap data", together with comparatively widespread structure sharing, would optimize for the wrong use cases

- Alternative II. Tracing garbage collection

  - Some complain about its unpredictability

  - Maybe harder to implement (need a runtime)

  - In any case unclear when to do sharing or not under the hood, given the desired ”by-whole-value” semantics, if we don't want as much sharing as in the ref-count model

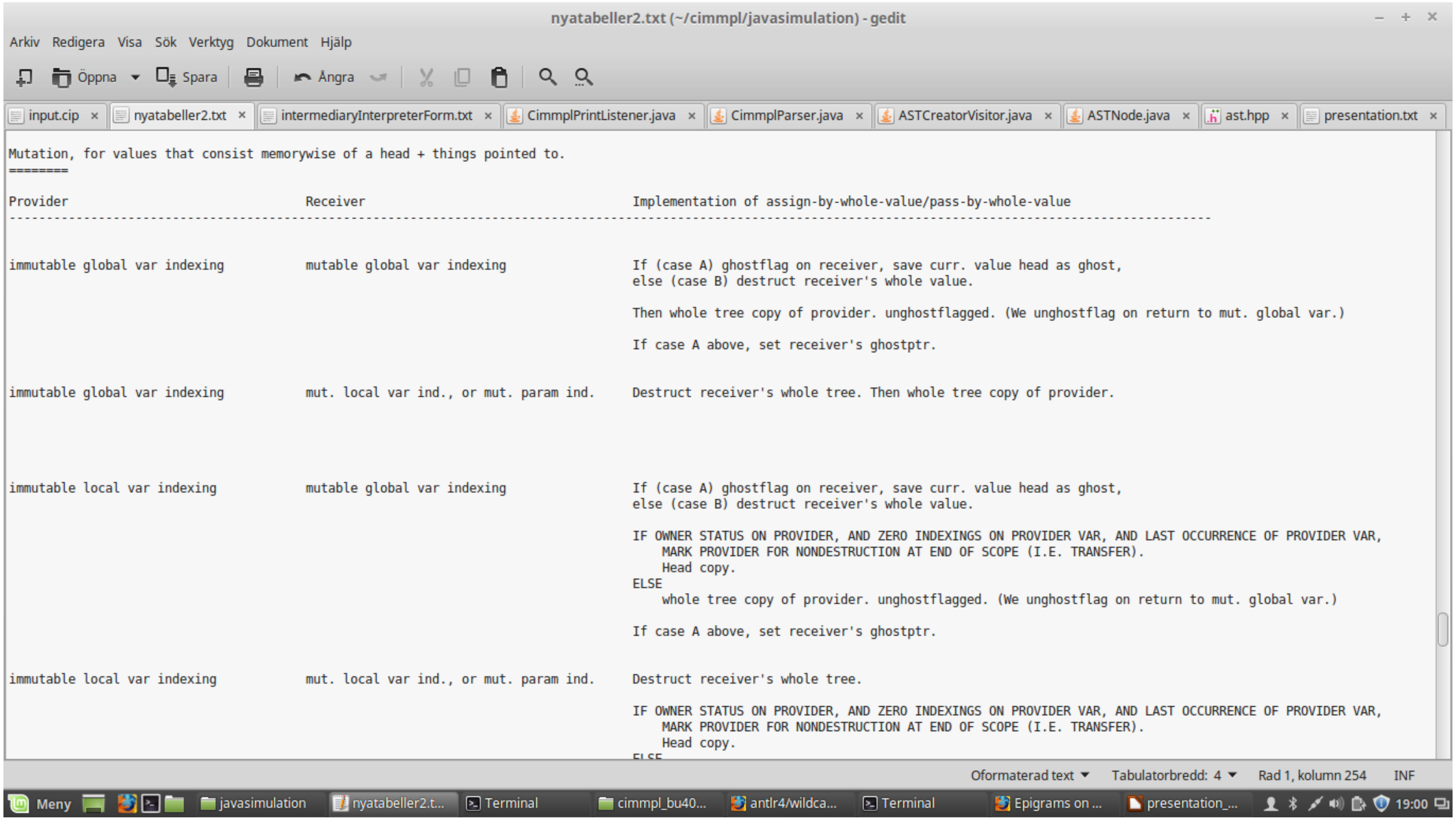
  - Why not do something different?

More alternatives?

- What do C/C++ programmers actually do?
  - ?... But they always want to know who **OWNS** something
- Problem is **much simpler** given Memtran's semantics

  In a language with the desired *”balloon pocket stack”* + *”assign/pass-by-whole-value”* semantics, no heap memory can become "free-floating” or shared -- everything has a unique owner naturally.
- But it would nice to have some modicum of speed, too!
- The language definition of Memtran (which had a different name back then) might perhaps be sufficiently simple that analyzing/optimizing case by case is doable?
- *I went through all the cases!*

The table


nyatabeller2.txt (~/cimmpl/javasimulation) - gedit
Arkiv Redigera Visa Sök Verktyg Dokument Hjälp
Öppna Spara Ångra
input.cip nyatabeller2.txt intermediaryInterpreterForm.txt CimmplPrintListener.java CimmplParser.java ASTCreatorVisitor.java ASTNode.java ast.hpp presentation.txt
Oformaterad text Tabulatorbredd: 4 Rad 1, kolumn 254 INF
Meny javasimulation nyatabeller2.t... Terminal cimmpl_bu40... antlr4/wildca... Terminal Epigrams on ... presentation_... 19:00

```
Mutation, for values that consist memorywise of a head + things pointed to.
========

Provider                                Receiver                                   Implementation of assign-by-whole-value/pass-by-whole-value
------------------------------------------------------------------------------------------------------------------------------------------------------------------


immutable global var indexing           mutable global var indexing                If (case A) ghostflag on receiver, save curr. value head as ghost,
                                                                                   else (case B) destruct receiver's whole value.

                                                                                   Then whole tree copy of provider. unghostflagged. (We unghostflag on return to mut. global var.)

                                                                                   If case A above, set receiver's ghostptr.


immutable global var indexing           mut. local var ind., or mut. param ind.    Destruct receiver's whole tree. Then whole tree copy of provider.




immutable local var indexing            mutable global var indexing                If (case A) ghostflag on receiver, save curr. value head as ghost,
                                                                                   else (case B) destruct receiver's whole value.

                                                                                   IF OWNER STATUS ON PROVIDER, AND ZERO INDEXINGS ON PROVIDER VAR, AND LAST OCCURRENCE OF PROVIDER VAR,
                                                                                       MARK PROVIDER FOR NONDESTRUCTION AT END OF SCOPE (I.E. TRANSFER).
                                                                                       Head copy.
                                                                                   ELSE
                                                                                       whole tree copy of provider. unghostflagged. (We unghostflag on return to mut. global var.)

                                                                                   If case A above, set receiver's ghostptr.


immutable local var indexing            mut. local var ind., or mut. param ind.    Destruct receiver's whole tree.

                                                                                   IF OWNER STATUS ON PROVIDER, AND ZERO INDEXINGS ON PROVIDER VAR, AND LAST OCCURRENCE OF PROVIDER VAR,
                                                                                       MARK PROVIDER FOR NONDESTRUCTION AT END OF SCOPE (I.E. TRANSFER).
                                                                                       Head copy.
                                                                                   ELSE
```

The table – Synopsis (1)

- Didn't get it completely ”right” at first attempt
- A value is always held by some **"holder"** (i.e. a variable instance or parameter instance, or a ”return operation”) in the program
- Important to distinguish **mutable** and **immutable** vars/params
- (If you group some cases together) there are seven kinds of holders that can be the holder of a value
- Initialization table has 49 cases, mutation table 14 cases

The table – Synopsis (2)

- *Return operations* are always **owners** (but they immediately transfer their ownership)
- Per-function "last occurrence" compile-time analysis shows when **transfer of ownership** is possible other than with return operations
- *Immutable parameters* mostly **non-owners**
  - But sometimes they should be **owners**
    - This must be known at call time.
      - Current solution: user can use keyword *"construand"* as an optimization directive
- *Immutable local variables* can be either **owners** or **non-owners**
  - Compiler figures it out at compile time, automatically
  - Compiler eliminates some local vars to facilitate transfer analysis (this can result in duplicated indexing operations, though)
- A special problem case is (borrowing from) *global mutable variables*
  - In the implementation, they can have **"ghost values"**
  - In order to deal with this, all heads pointing to heap data carry an extra flag and a pointer
- No extra annotations on "data part" of values

Reasoning

- **Immutable params** (of global functions) can be **borrowers**, because:
  - If **provider is a local var/param**, then that provider ***cannot be accessed*** for modification during function execution.
  - If **provider is a global var**, fallback is provided by the ***ghost system.***
    (A bit complex: ghost *”subvalues”* can have ghosts, whose *”subvalues”* can have their own ghosts...)
  - If **provider is a return operation**, we can still borrow the value if ***compiler transforms*** so that the value has a ***temporary owner*** in the calling scope, during function execution.
- But some immutable params should be **owners**
  - Pragmatic solution: Invent the concept **”construand”** for explaining the distinction in language-semantic terms, and let user provide it manually
- **Per-function analysis** can deduce when **transfer** is possible (we don't do it for global vars/params)
- Immutable variables can be **sharers** if an immutable var/param is the provider

Some evaluation

NOTE: THIS SLIDE IS COMPLETELY BOGUS AS ALL THIS IS NOT IMPLEMENTED YET AT ALL

- Runtime behaviour is more or less predictable (in theory)
- But the table can be hard to memorize
  - It's possible to give a simplified summary though
- Sometimes the implementation copies whole values where hand-optimized C code would not
- Is it OK?
- Too early to say...
- Maybe we can add something to the language improving the situation, in case it's needed...

Why the ”ghost system” is needed

```
mu data ' arr int = [1, 2, 3]

/* Adds reverse of input vector to global vector 'data' */
fn manipulate_data(input ' arr int) :
    for it over data,
        jt in input indexfactor -1 indexoffset len(input) - 1
    :
        it += jt
    ;
;

manipulate_data(data)
print(data)    // should print 4 4 4
               // might print 4 4 7 without ghost system
               // and the language designer has decided to allow cases
               // like this, because given the
               // official rendition of the language's semantics,
               // it would be hard to explain why it would be forbidden
```

Features that do not easily fit into the model

(and which have thus been left out)

- Exceptions
- True closures
  - but we ***do*** have first class functions and even local functions

    (just with a few restrictions...)
- Currying
- Point-free style
- Call by current continuation

The for loop

- Starts with an increasing/decreasing variable range**.** This is the foundation for the iterations over the arrays, if any.
- The variable range can be left out, though -- then implicitly defined as range of first array
- Then follows iterations over arrays using **over** or **in**
- **over** is for mutation – iteration variable is not a real variable but an ”alias”; array lvalue refers to a slot or subslot – i.e. cannot be any expression
- **over** and **in** iterations can have an **indexfactor** and/or an **indexoffset**
- The start and end points of iterations are calculated and boundschecked, before entering loop body
- As long as you don't do explicit indexing with [ ] in loop body, no bounds checking is done during looping

Thank you for your interest!!!

Further documentation:

http://github.../manual.html

GITHUB: http://www.github.etc

Comments and questions welcome at:

mani@memtran.org