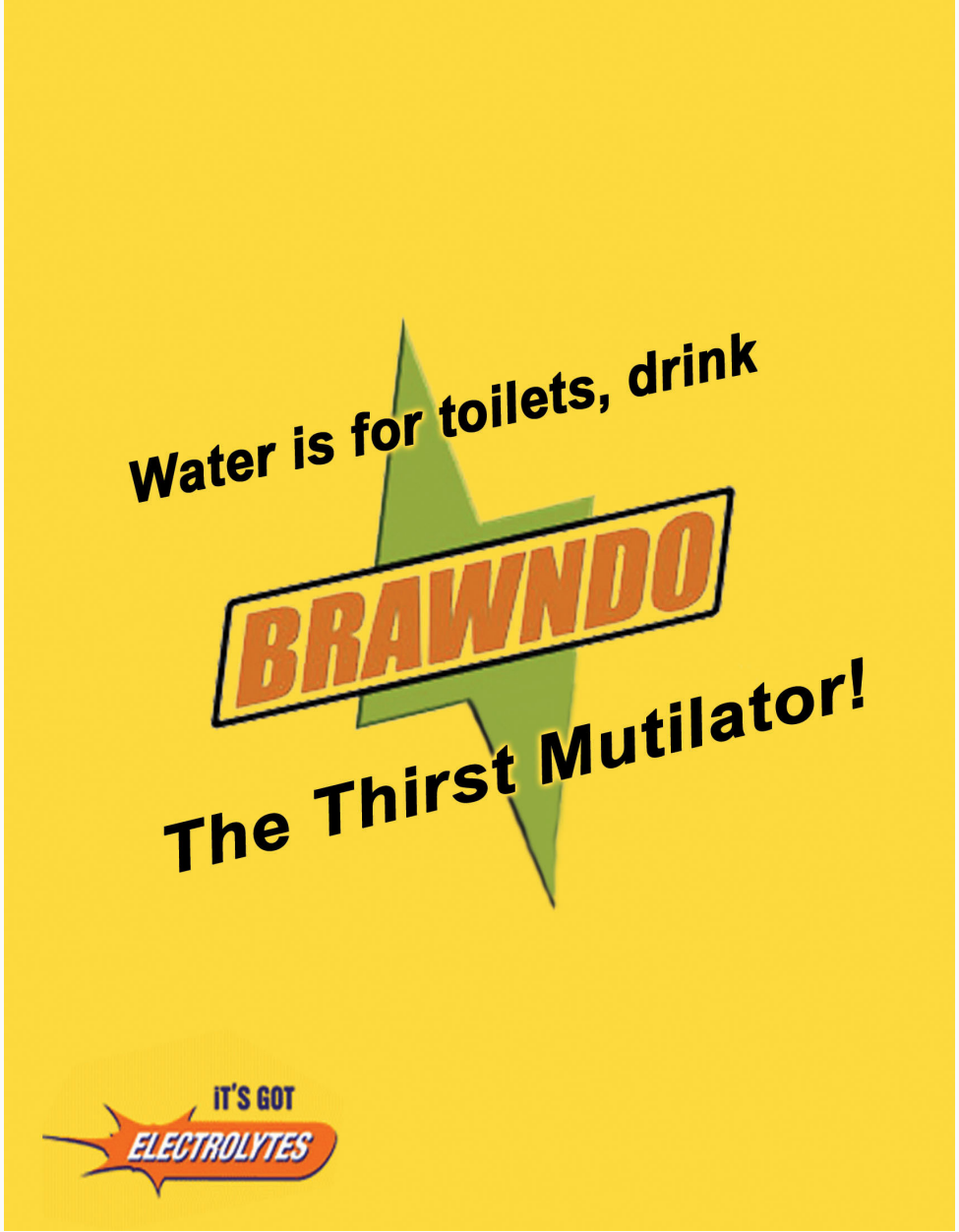
Using Golog (Prolog interpretator in Go)

Oleg Sivokon

<2017-05-01 Mon>

Outline

Water, From The Toilet?


Water is for toilets, drink
BRAWNDO
The Thirst Mutilator!
IT'S GOT
ELECTROLYTES

Various models of computation, unknown to general public

- Petri nets a.k.a. Turing machine
- “Value semantics”, a.k.a. “functional programming” a.k.a. “declarative programming”
- Unification
- Graph rewriting
- Cellular automata
- Biological modes? Perhaps much, much more. . .

Programming taught using one model

- Model intuitively accepted as the only possible one
- Students don't develop more abstract understanding of computation
- Some problems lend themselvs better to particular computation model

Serialization

> *This comes up when you're doing generic programming, where you want to write functions that will take data of any type and just walk over it and serialize it, say. So that's a time when types can be a bit awkward and an untyped language is particularly straightforward. It couldn't be easier to write a serializer in an untyped language.*

Simon Peyton Jones, GHC author, Microsoft Research.

Parsing

> *Prolog was originally invented as a programming language in which to write natural language applications, and thus Prolog is a very elegant language for expressing grammars. Prolog even has a builtin syntax especially created for writing grammars. It is often said that with Prolog one gets a builtin parser for free.*

Prof. David S. Warren, University of New York

Homoiconicity

> *The ability to represent the code in the language by other code in that language makes that language deserve the status of homo-iconic. Since version 3.0 C# was homoiconic for the expression subset of the language, thanks to the introduction of expression trees. As you can see, the expression trees are a very powerful concept to inspect code at runtime which goes much beyond their use in LINQ. You can think of them as "reflection on steroids", and their importance will only increase in the releases to come.*

Bart De Smet, C# 5.0.

Some practical uses

- Knowledge bases
- Interactive shells
- Rich data exchange and configuration format
- Testing

Unification

```
%% Two uninstantiated variables unify
X = Y,
%% Unification works on sub-terms
%% X = 1
[X | Xs] = [1 | Xs],
%% Unification can be arbitrary deep
%% and unify multiple variables
%% X = 1, Y = 2, Z = baz(3)
foo([bar(X, Y), Z]) = foo([bar(1, 2), baz(3)]),
%% Unification can be selfreferential
%% X = rec(rec(rec(...)))
X = rec(X)
```

Suggested literature

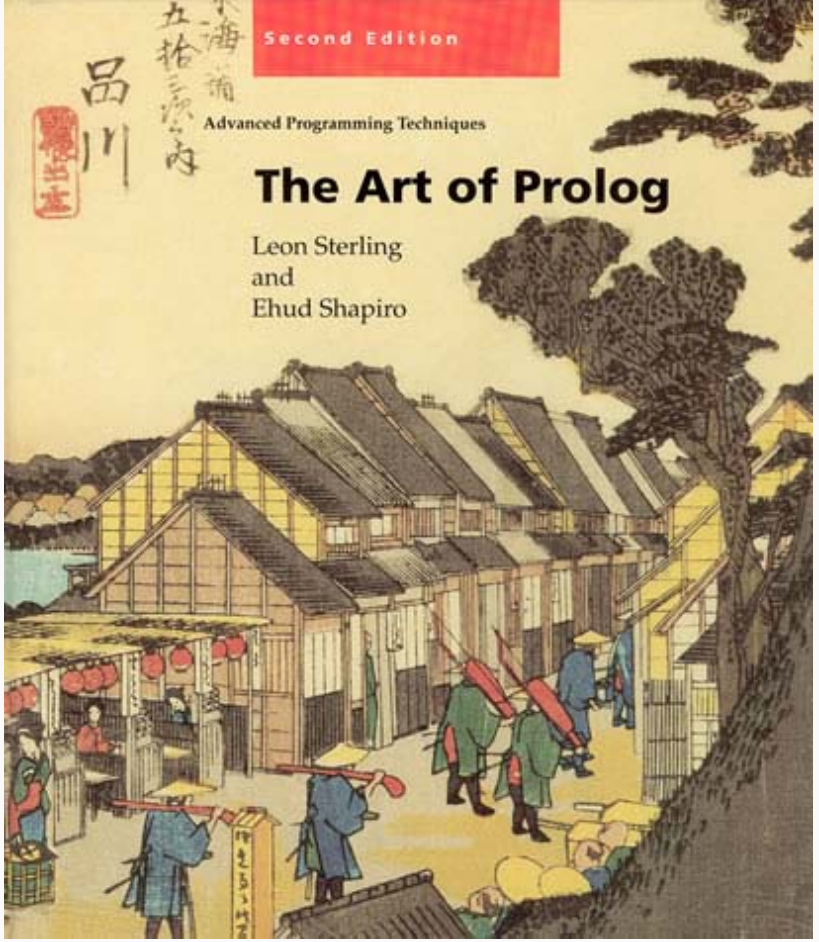


Figure 1: `https://mitpress.mit.edu/books/art-prolog`

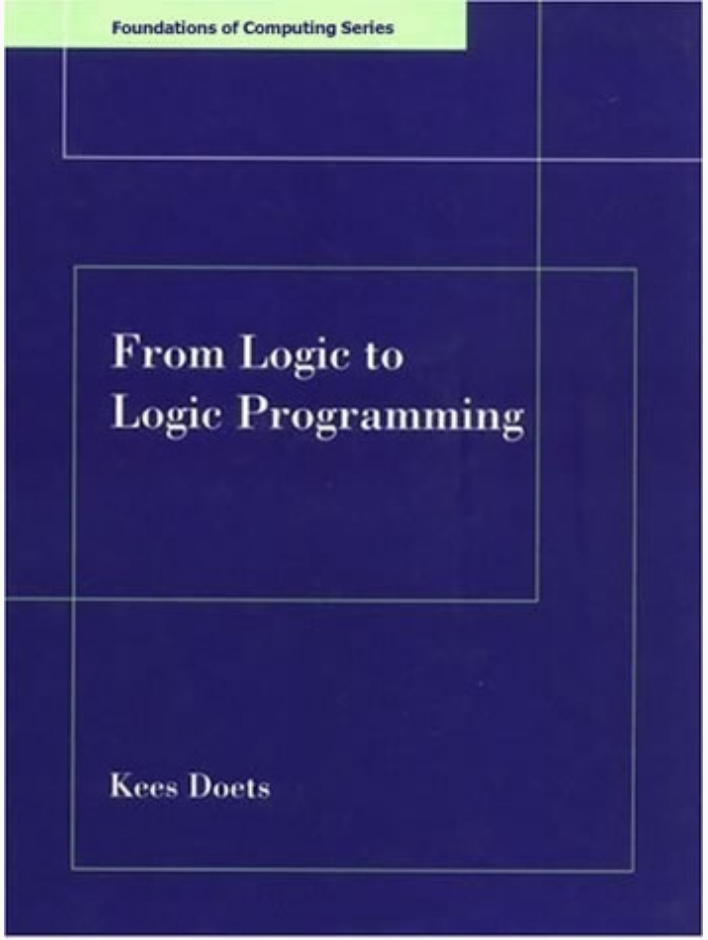
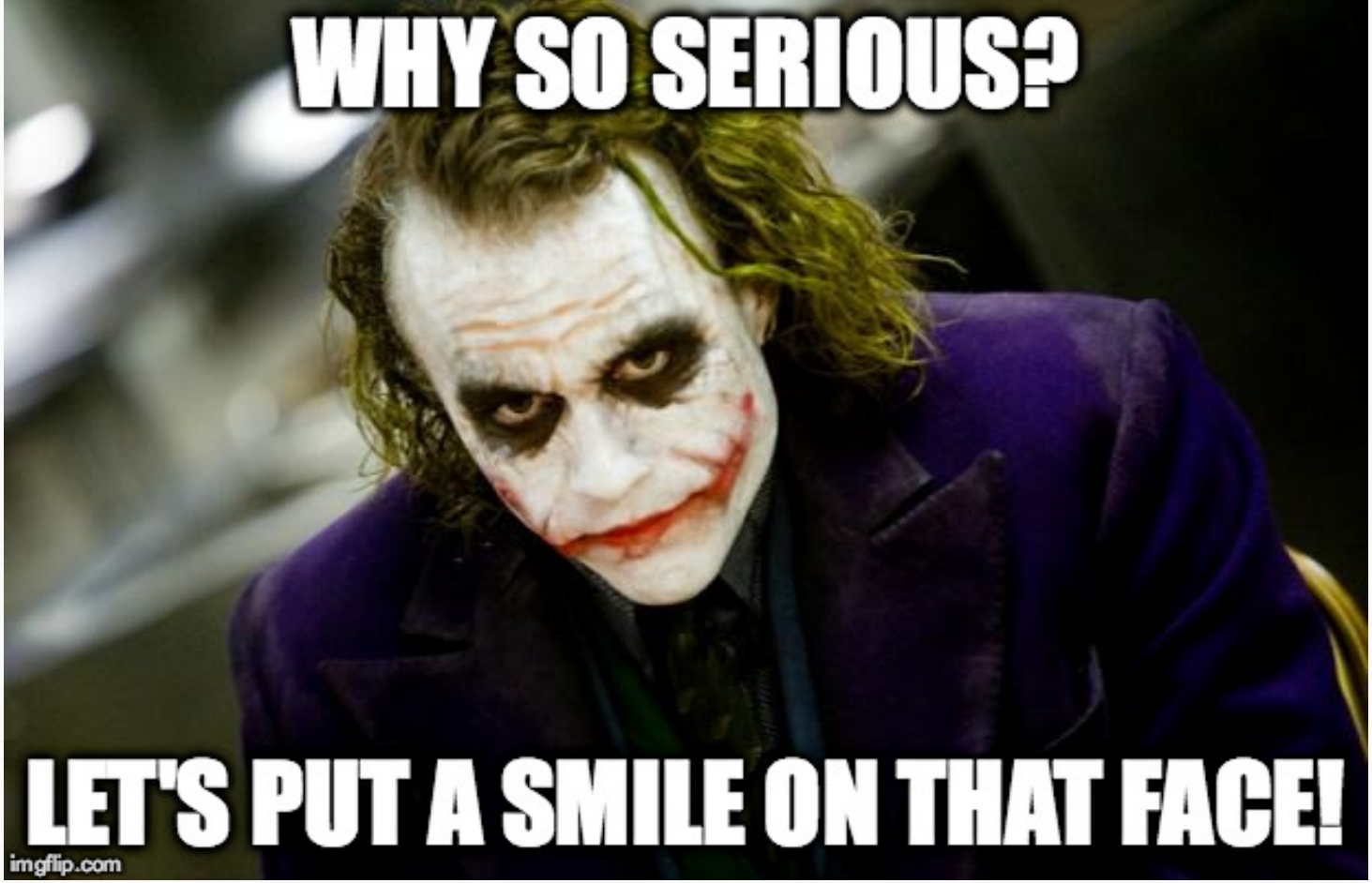


Figure 2:

`https://mitpress.mit.edu/books/logic-logic-programming`

Gentle Introduction


WHY SO SERIOUS?
LET'S PUT A SMILE ON THAT FACE!
imgflip.com

Obtaining the code

- Official version: git@github.com:mndrix/golog.git
- My fork: git@github.com:wvxvw/golog.git

Minimal setup

```
m := golog.NewInteractiveMachine()
m = m.Consult(`
    father(john).
    father(jacob).
    mother(sue).
    parent(X) :- father(X).
    parent(X) :- mother(X).
`)
solutions := m.ProveAll(`parent(X).`)
for _, s := range solutions {
    fmt.Printf("%s is a parent\n", s.ByName_("X"))
}
```

golog.Machine

- Main interface to interpreter
- Immutable *(safe for concurrent access)*

(*golog.Machine).Consult(interface{}) *golog.Machine

- `consult/1` loads code in ISO Prolog
- Creates database
- Similar to `insert` in SQL

(*golog.Machine).ProveAll(interface{}) []golog.Bindings

- Generates ALL solutions
- Similar to `select` in SQL

Exposing foreign (native) objects

```
func HelloWorld(
    m golog.Machine, args []term.Term,
) golog.ForeignReturn {
    return golog.ForeignUnify(
        term.NewAtom("Hello World"), args[0],
    )
}
func NewHelloWorldMachine() golog.Machine {
    return golog.NewMachine().RegisterForeign(
        map[string]golog.ForeignPredicate{
            "hello_world/1": HelloWorld,
        },
    )
}
```

Using foreign objects

```
m := golog.NewHelloWorldMachine()
solutions := m.ProveAll(‘hello_world(X).‘)
for _, s := range solutions {
    fmt.Printf("%s\n", s.ByName_("X")) // Hello World
}
```

Marshalling Go structs (experimental)

- `native.Decoder` decodes Prolog callable terms into Go structs
- `native.Encoder` encodes Go structs as Prolog terms
- `native.GenerateMethods()` generates methods specializing on given struct
- `native.GenerateAccessors()` generates special Prolog predicates for accessing fields in Go structs
- Limitations: Go methods don't backtrack. Some reasonable functionality is still to be implemented.

Extending

- Example: `term.Native` type
- `golog.RegisterForeign` to create machines with special set of predicates
- `golog.Machine` is an *interface!* Possible other implementations include:
  - Relational database access
  - Distributed machines
  - Concurrent machines

Writing A Build System Using Golog

Sources

- Example project: `https://github.com/wvxvw/logomake`

Writing Rules

```
hello_world :-
    c(["hello_world.c"], "hello_world").

all :-
    hello_world.
```

Slightly Advanced Rules

```
not_tests(Sources) :-
    findall(CFile, (
                glob("./*.c", CFiles),
                member(CFile, CFiles),
                \+ prefix(CFile, "test_")
            ),
            Sources).

all :-
    not_tests(Sources),
    printf('Building %s~n', Sources),
    c(Sources, "program").
```

Calling From Command Line

```
logomake [--makefile <file>] \
         [--goal <prolog term>]
```

- Default file is `Makefile.logomake`.
- Default goal is `all.`.
- Allows evaluating arbtirary Prolog goal against the database in the Makefile.
- With a slight change, can run interactively.