JOHN - A Knowledge Representation Language

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Research Note RN-2008-003

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

JOHN - A Knowledge Representation Language

Main » Knowledge Representation Model

Research Note: Lessons Learned (so far)

One: The Best Ideas in Your Work End up Coming from Your Colleagues.

Only if you:

- Be working with a greatest group of Computer Scientists in town.
- Keep them engaged in your work and
- Discuss your thoughts and problems with different colleagues as much as near the boarder of being annoying.

Two: Is There Common Ground between Natural and Machine Language?

A middle ground between what appears like natural language, understandable by an average non-programmer person, yet unambiguous and well defined, understandable by machines is still hard to find today. An average person is afraid of working with computers for any not-so-common task because it needs to be "programmed." Why should the average fellow know what an *array* or *pointer* is to be able to program a computer for a simple common task?

The majority of our world knowledge is common sense. Object oriented programming methodologies tend to be the natural way of our understanding of the world around us. Message passing style provides us a mechanism to construct natural language-like expressions that yet are unambiguous and understandable by a machine:

```
   English: "A dealer can deal if it has at least 5 cards"
      JOHN:  "rule Dealer deal if its cards size >= 5."
   English: "Robot: please pick all red tall blocks"
      JOHN: "Robot do pick (all red color tall size Block).
   English: "Are all the blocks on the table?"
      JOHN: is for every Block do each position = the Table?
   English: "A List is ordered if its size is less than two or
             its first element is less than the second and the list starting from its second
             element is also sorted."
   Spanish: "Una lista esta ordenada si su tamaño es menos que dos, o
             si su primer elemento es menos que su segundo y la lista a partir de su segundo elemento es también ord
   EL JOHN: "calificar Lista ordenada si su tamaño <= 2 o
             su primero <= su segundo y su resto ordenada = si."
Portuguese: "..."
      JOHN: "sorry, this language isn't supported."
```

Three: The Goodness of Using Real COLA Objects - Actions and Goals - Methods - Variable Bindings

In my first attempt of writing the interpreter in COLA language , I synthesized objects and actions by keeping a dictionary of names and associated properties and expressions. In second pass, I used real Cola objects as the Micro-world Objects. The classes are defined as real COLA classes (types). The qualifications (methods w/o side-effects - just functions) and actions (state changing methods) are real methods those objects understand. Goals are also methods which launch a search for a particular state. Among the many advantage of this alternative are:

- Binding of arguments for methods are automatically done by COLA
- All the internal messages understood by COLA objects such as *<collection> size*, etc are automatically supported

- Implementing inheritance is a freebie
- and many more.

As a result the lines of code relative to the first version of JOHN was cut down by almost half, while the speed increased by ~50-60 times.

Four: All About the Awesomeness of Possible Worlds

In my first implementation of JOHN in Lisp , I used a painful step of caching the changes and undoing those changes in reverse order to implement search of possible future scenarios. In the COLA implementation, I implemented the state of the micro-world as a world, and search as looking at possible worlds. A world is a particular state of the micro-world, holding the property values for objects at some instance in time. Possible Worlds provides an elegant, incredibly simplifying, life changing methodology for:

- There is absolutely no need for backtracking, undoing. Possible worlds are sprouted by the parent world and they all coexist. Once a desired world is found, the micro-world is "magically" moved to that world by a simple *currentWorld* pointer update from the old world to that desired world. Going back to any past states is equally as easy as changing this pointer to some past world.

- Worlds are time stamped and with every change, the micro-world moves to a new world. Thus at any point of time, the state of an object at any past time can be queried. Only changes relative to the current world would go into the child world (world at time `t+1`), thus all lookups are dynamically scoped. Each world has a pointer to parent, as well as any number of children.

- At any point of time, the possible states of some object at some future time can be deduced. This is done by getting a list of all possible changes that can occur to the objects (actions or methods that each object is capable of) with every combination of arguments for those actions, thus generating a set of possible worlds via running those actions each in a new child world.

```
-----------------
|   |   |   | G | 4          R: Robot position
-----------------
| R |   |   | X | 3          G: Goal Square
-----------------
| . |   |   | X | 2          X: Blocked Square
-----------------
| . |   | X | X | 1          .: Visited (also Blocked)
-----------------
  A   B   C   D
```

Assuming R is a Robot that is capable of moving one square `Up`, `Down`, `Left`, or `Right`, provided the target square is not blocked (X marked), nor a previously visited square:

Assuming the current time of the Maze micro-world is 2, we can query the position of the robot at some point of time in the past. Here the position property of a Robots object called `Robot` is obtained at a world with time stamp of 1:

```
> time.
2
> Robot position.
A3
> at time 1 Robot position.
A2
```

```
world w0     world w1    world w2
time 0       time 1      time 2
----------- ----------- -----------
| w0: t=0 | | w1: t=1 | | w2: t=2 |
| R @ A1  | | R @ A2  | | R @ A3  |
----------- ----------- -----------
```

Still assuming we're currently at time 3, we can query a property of an object in some time in the future. In that case, the interpreter looks at all possible actions doable by that object to produce a set of all possible world values that this property may have at the given future time:

```
> at time 3 Robot position.
Possible World values (1 time unit from now):
{ B3, A4 }
```

```
                                                    time 3
                                                  ----------
       time 2        / Robot move Up? ---------> | w3a: t=3 |
     -----------    /  ___ Robot move Down? No.  | R @ A4   |
     | w2: t=2 |   /__/                           ----------
     |  R @ A3 |  //
     -----------   -------- Robot move Left? No.
                   \                              ------------
                    \ Robot move Right? --------> | w3b: t=3 |
                                                  | R @ B3   |
                                                  ------------
```

Similarly, building the possible worlds tree for even farther time ahead:

```
> at time 5 Robot position
Possible World values (3 time unit from now):
{ B3, A4 , C4, C2, B1 }
```

Another advantage of worlds is that since any object properties are static in a given world, all properties and relation lookups can be cached. Also the children of a world (possible worlds of time `t + t0`) are never calculated twice, since the worlds cache their parent and children worlds.

Moving from parent world to a child, the child world stores the action taken in the *causingAction* field. This field along with the *parent* field makes it possible to get the series of actions (solution) in the path from the initial world to the current world.

Five: Goal Oriented Programming

The micro-world programming model in JOHN is like this:

- Define the classes involved in the micro-world and the properties that matter for each.
- Instantiate as many objects as needed belonging to any one of those classes that were previously defined, along with their initial property values.
- Define how the change can come about in this micro-world via actions that the objects are capable of doing.
- Define goals for one or more classes in the micro-world, that are achievable by some sequence of actions previously defined. The goals are simply predicates stating a desired state for the object.
- Heuristics and Optimizations may be defined that help the object find their goal states faster.
    - Goal Heuristics give insights to which of alternative choices (of possible worlds) in the search tree are more likely to get us to the goal faster.
    - Optimizations define which actions, and/or with which action arguments should best suit getting the object to its goal state.

It turns out that many of typical programming task can be viewed and achieved with a goal-oriented mind. Sorting for instance, can be viewed as a *be-sorted* goal belonging to the *List* class, with potentially many candidate procedures (actions) that may achieve it, such as *merge-sort*, *bubble-sort*, etc. Flexible, dynamic optimizations may be defined that tune the right mix of procedures depending the nature of the list, considering its size, distance of items in it, etc.

Six: Meaning vs Optimizations

There are two main parts to any algorithm: the invariant (meaning) is what we're trying to accomplish, and the procedures (optimizations) which are ways to accomplish it.

Any kind of sort program's invariant is equal, basically describing what it means for a list to be sorted. this will never change. Sort programs also contain one or many procedures that achieve the goal of sorting the given list. The problem with most of the programs today is that it's hard to distinguish the two main parts, even hard to tell different procedures apart, even harder to control which combination of procedures to try in the presence of multiple ones. Another setback is that the programs are often not understandable for the same reason. A disorganized amalgam of meanings and optimizations are messy, hard to understand and control.

JOHN programs separate the meaning and optimizations by staying mainly declarative. Meanings are stated in forms of *goals* possessed by objects. Procedures appear as *actions*, by some sequence of which these goals can be achieved. They themselves are defined separately and distinguishable from one another. Actions by themselves are not smart to know what their purpose in life is, or when is that purpose achieved. It is the meaning part, the goal, that watches these actions done and takes over when it is satisfied (or unsatisfied).

```
; AList is a type whose items field is its original list and its sorted-items field starts empty and will be the so
create AList items sorted-items.
```

The meaning of sort:

```
qualify List sorted if its size < 2.
qualify List sorted if its first <= its second and its rest sorted = yes.
; assuming permutation-of relation is also defined beforehand...
goal AList sort try its sorted-items permutation-of its items and
                    its sorted-items sorted = yes.
```

The sort procedures:

```
action AList selection-sort consequence ...
action AList quick-sort consequence ...
action AList pop-min consequence ...
action AList swap-items consequence ...
action AList broken-sort consequence ...
```

The good thing about the methodology is that user can explicitly turn any combination of these procedures *on* or *off* in achieving a certain goal.

Explicitly picking one procedure to do the sorting:

```
> L1 satisfy sort using quick-sort.
```

Explicitly picking more than one procedure to do the sorting:

```
> L1 satisfy sort using quick-sort broken-sort.
```

Seven: Goal Optimizations

More realistically, we like to dynamically decide the mixture of procedures to use to accomplish the task at hand. JOHN allows optimization definitions for goal to achieve this.

The sort optimization control:

```
goal-optimization AList sort use swap-items if its items almost-sorted = yes.
goal-optimization AList sort use pop-min if its items almost-sorted = no.
```

Sample Run 1: for list L1 the optimizations choose "swap-items" procedure

```
> make AList L1 [ 3, 1, 4, 5 ] [].
ok.
> L1 satisfy sort by L1.
looking 1 time unit ahead (1 possible worlds)
satisfied at time: 1
satisfied by: [ L1 swap-items  ]
```

Sample Run 2: for list L2 the optimizations choose 4 iteration of "pop-min" procedure as the best way to sort

```
> make AList L2 [ 4, 3, 5, 1 ] [].
> L2 satisfy sort by L2.
looking 1 time unit ahead (1 possible worlds)
looking 2 time unit ahead (1 possible worlds)
looking 3 time unit ahead (1 possible worlds)
looking 4 time unit ahead (1 possible worlds)
satisfied at time: 4
satisfied by: [ L2 pop-min , L2 pop-min , L2 pop-min , L2 pop-min  ]
```

It is also possible for optimizations to choose a mixture of procedures, as long as those procedures can be used collaboratively to get to a goal state. In the example below, we are trying to get the first 10 numbers in the Fibonacci sequence.

We like to use the nice mathematical formula `Fib(n) = Fib(n-1) + Fib(n-2) , for all n > 1` as long as the numbers are small, but this is a rather expensive calculation and after a while (say after `n>5`) we like to use the iterative method to get the next fib number in the sequence, by simply adding the last two items in the sequence to get the next Fibonacci number. We call the two procedures `slow-fib` and `fast-fib`.

Optimizations, assuming "fib-sequence" goal for class "Calc" already defined:

```
goal-optimization Calculator fibonacci-sequence use fast-fib if its answer size > 5.
goal-optimization Calculator fibonacci-sequence use slow-fib if its answer size <= 5.
```

Sample Run: The object has used the slow-fib for the first 4 fib numbers, and then fast-fib for the rest

```
> Calc answer.
[ 1, 1 ]
> Calc satisfy fibonacci-sequence 10.
satisfied at time: 9
satisfied by: [ Calc slow-fib , Calc slow-fib , Calc slow-fib , Calc slow-fib , Calc fast-fib , Calc fast-fib , Cal
> Calc answer.
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ]
```

Eight: Action Optimizations

When accomplishing a goal, we like to use optimizations to choose the best combination of procedures to achieve the task of satisfying the goal. Equally, procedures may take arguments and themselves may have optimizations for choosing the right value for those arguments. In other words, the procedure need not to be called with explicit arguments, but the most suitable arguments may be deduced by a set of optimizations.

As mentioned, the set of next possible worlds given a world is generated by trying all actions (procedures) with every combination of parameters they may take. We can define optimizations for any of these procedures that can build the list of argument combination dynamically.

For instance in the chess program, a `Player` object may be capable of doing a `move` action that takes two parameters: the `From` square and `To` square:

```
          action Player move Square:From Square:To consequence ...
```

Thus the interpreter will try all permutations of a pair of chess board square objects (64 * 64 of them) to deduce what changes the player may do to the board on its next move. However, most of such permutations are not valid chess moves. Although the rules defined for the move action will catch them and simply discard such moves, it is really inefficient to try that many actions. We like to know better and not even try the moves that don't make sense.

One first hand method to do this is defining an optimization for this action that cuts down the possible action list by saying the `To` variable has to be `in-chess-move-span` of the From variable, assuming such relation is already defined as the two squares being either on the same row, same column, same diagonal, or a knight move apart with respect to one another (this covers all pieces moves):

```
          action-optimization Player move establish To in-chess-move-span From.
```

The result of such optimization is cutting down the 4096 possible parameter permutations to 1856, hence avoiding (4096 - 1856) method calls (that we knew beforehand were going to fail anyway).

Nine: Heuristics

It is often the case that the possible worlds tree of some micro-world is too large for a brute force search. If the micro-world being described happens to be a game like chess, it is surely impractical to search beyond a few levels deep in the tree. Search heuristics are thus used to effectively prune the search tree by assigning a score to a set of possible worlds and picking the highest scoring one (or ones) as the only possibility.

Heuristics also go along with our intention to keep the parts of the program isolated, extensible, and understandable. Any number of them can be added or removed, effectively increasing the level of both intelligence and efficiency of the program.

Going back to the chess example, note that humans don't really play chess the way machines do: search of all possibilities. We usually have certain "strategies" in the back of our heads, such as "capture pieces when you can," "avoid being captured," "try to dominate the board with your pieces...", etc. Looking at the board at any given time, we make a move having one or some of those strategies in mind.

The chess JOHN program also goes about making moves in a similar fashion. Below are three heuristic we added effectively implementing the strategies mentioned above. Nowhere else in the chess program but the heuristic section you may find a chess strategy/optimization. Any more number of heuristics may be added here that can make the program a stronger player, just as a novice player learns more and more strategies as she becomes a better player.

In the presence of multiple heuristics, the cumulative score for all of them is considered. Thus the weights given to the heuristics allow for prioritizing the heuristics. In chess, all pieces have a `worth` property, signifying their value relative to other kinds of pieces. Hence summing the worth of each players pieces constitutes an accurate measure of the player's piece strength. Here we're teaching the computer that capturing pieces is more important that having pieces in strategic locations:

```
          heuristic Player win 1000 maximize its pieces worth sum.
          heuristic Player win 1000 minimize its opponent pieces worth sum.
          heuristic Player win 1 maximize its centralized-pieces size.
```

Ten: Heuristic based Search Tree Pruning (or Branch Re-Ordering)

Most heuristic based search methods like *A** use the given heuristic at every search tree level to reorder the *open list* (possible worlds) to explore. I experimented with a slightly different approach to implement the search, which has definite disadvantages as well as some nice properties.

I allow the user to set a parameter called `max-time-ahead` which tells how many levels of the search tree should we explore, before using any heuristics to give those possible worlds their scores. For instance, going back to the maze example we may have the following scenario.

> max-time-ahead = 3: effectively means the robot has a "flashlight" lighting up 3 blocks ahead, leaving the rest of the blocks yet "dark" and unseen. The unshown blocks are yet "unseen" by the robot. The ? symbols show where it may be at time = 3:

```
      -----                                             possible worlds tree    time
      | ? |           4                                        A1                0
      ---------                                              /    \
      |   | ? |       3                                    A2      B1            1
      ---------                                           /  \        \
      | ? |   | ? |   2                                 A3    B2       B2        2
      -------------                                     /\   / | \     /\
      | R | ? | X |   1                              ----==-----------------
      -------------       possible worlds --->      | A4 B3  B1 C2 B1  A2 C2 |   3
        A   B   C             of t = 3               ----==-----------------
                                                          |
> at time 3 Robot position.                               | say B3 is picked
Possible World values (3 time unit from now):             |  by heuristics
{ B1, A2 , A4, C2, B3 }                                   V
                                                            pruned tree
                                                        (move A1 to A2 is certain)
                            ^                                      A1
                            |                                     /
                            |                                   A2
                                                              /   \
                     Repeat above by         <----------     A3    B2
                 getting possible worlds                    / \   / | \
                        of t = 6                         A4 B3 B1 C2 B1
                  (and t = 9, so on...)
```

Now the robot realizes it will not reach the goal before this time and has to use the heuristics to decide which one of the 5 possible worlds it best likes to be at time `t= 3`. What happens next is that only the tree branch with the root node at time `currTime + 1`, which includes the picked possible world is kept, and other branches are pruned. The search with the new pruned tree is repeated now with parameter `max-time-ahead` increased by itself, effectively searching up to `t = 6`, and the next time up to `t = 9`, and so on. This continues until the goal is satisfied, or no more possible worlds can be generated, the case of unsatisfiable goal.

The disadvantage of the current implementation is that the search tree is *pruned* instead of *reordered*. This makes the algorithm an *incomplete* one, which means may not find an existing solution because of a heuristic leading it into a trap. This will be implemented in the later versions.

There can be a trade off between heuristics and search speed. The advantage of this method is that it provides the user the flexibility of how much of heuristic evaluation should be done. While most heuristic based searches use `max-time-ahead = 1` (follow heuristics at every search level), sometimes in the presence of many heuristics with very large possible worlds trees, the heuristic evaluations themselves may slow down the search. Our method gives more control over the amount of heuristic usage. Also this method still guarantees the found solution is optimum, by effectively implementing an Iterative Deepening Depth First Search (IDDFS).

Eleven: Goal-Oriented Actions - Using Search and Heuristics to Do Actions

Actions are implemented as *methods*, and as methods they may take arguments. It is possible in the JOHN interpreter to run an action without providing the arguments, which means it should find the right arguments to call based on some defined goal of the object and search among possible worlds. Once again, if the goal is not reached within the `max-time-ahead` limit, the heuristics will kick in and pick the best possible world, and the next action in the picked possible world's path will be taken.

Twelve: Implementing Two Player Games

When the micro-world being described is a two player game, the `two-player-game` option is used. The value of this option is set to a list of the two opponents objects, assuming such objects are already defined:

```
          > set-opt two-player-game [ White, Black ].
```

The effect of setting such option is that the search algorithm will go into the Minimax mode. In Minimax mode, while still all possible moves for the player are included in the search tree, only the opponent's best moves based on heuristic scores are included.

Thirteen: More Late-bound than As-late-bound-as-it-gets?! - Circular Definitions

It is a common need when describing micro-worlds? for two or more objects have each other defined for properties. For instance, assume we're implementing a two player game, where the Player class has two properties called opponent and turn. the opponent field is set to the player's opponent object.

```
          > create Player opponent turn.  ;; defining Player class
          ok.
          > make Player White Black yes.  ;; making a new Player object called White
          ok.
          > make Player Black White no.   ;; making a new Player object called Black
          ok.
          > White opponent.
          Black
          > White opponent turn.
          no
```

Circular definitions are supported by the interpreter: In the example above although Black object is not defined yet, White object's opponent property can point to it. As soon as the Black object is defined later, the White's opponent field updated accordingly to point to the newly defined object.

To support this feature a hash table of referenced undefined objects and a dependency list for each is created, and properties are updated as soon as any of those objects becomes defined.

Fourteen: Referencing Object via Expressions

Objects can query other objects using select expressions, but it may be inefficient. For instance in an NxN? grid of block objects, each object may know its top neighbor by.

```
          > create block col row.
          ok.
          > make block A1 A 1.
          ok.
          .
          .
          .
          > all block.
          { A1, A2, A3, B1, B2, B3, C1, C2, C3 }
          > qualify block top-neighbor one-from select all block by which row = its row - 1 and
                                                                     which col = its col.
          ok.
```

```
> B2 top-neighbor.
B3
```

Which is an O(N^2) search of all block objects. However, if we have prior knowledge of how these block objects are named, it is possible this query in `constant` time:

```
qualify block top-neighbor (its name first + (its row + 1) ) object.
```

The `<name> object` command returns the named object. This is possible because the system keeps a dictionary of instantiated `objects` with their names as keys.

Another example:

```
> White turn.
yes.
> ("Whi" + "te") object turn.
yes.
```

Fifteen: Internal Data Structures of JOHN

There are two main components of a micro-world. The static part (definitions - knowledge base), and dynamic parts (state or world - working memory).

```
 Knowledge Base                                Working Memory
 (statics)                                     (dynamics)
 ____________________________________         __________________________________________________
|                                    |        |                                                |
| Microworlds                        |        | currentMicroWorld                              |
| ___________________________________|        | currentWorld                                   |
|| Microworld                        |        | Worlds                                         |
|| __________________________________|        | _______________________________________________|
||| Classes     Instances            |        || World                                         |
||| _________________________________|        || ______________________________________________|
|||| Class                           |        ||| time                                         |
|||| ________________________________|        ||| parent                                       |
||||| propertyNames                  |        ||| children                                     |
||||| instances                      |        ||| causingAction                                |
||||| Actions         Goals          |        ||| objectsPropertiesValues                      |
||||| _____________   ______________ |        ||| (dictionary: objectName X                    |
|||||| Action      | | Goal         ||        |||             ( propertyName X propertyValue )|
|||||| ___________ | | ____________ ||        |||_____________________________________________|
||||||| argNames  || || argNames   |||
||||||| argClasses|| || argClasses |||
|||||||           || || heuristics |||
|||||| ------------  | -------------||
||||| --------------  ---------------|
|||| --------------------------------|
|| ----------------------------------|
```

Sixteen: The Beauty of Allowing Language Extensions to Be Written in Itself

My original thought was for the micro-world description language to be plugged into some external graphics software to visualize the micro-worlds. One of my colleagues suggested using the language itself to describe the graphical properties of objects. This idea turned out to be a powerful one, allowing me to implement a playable chess program that includes ASCII graphics like below in an around 250 lines:

```
-----------------------------------------------------------------------------------------------------------------
|    {===}    |.   {===} .  |    {===}    |.   {===} .  |    {===}    |.   {===} .  |    {===}    |.   {===} .  |
|     |=|     |. .  |\)  . .|     \=/     |. .  |=|  . .|     |=|     |. .  \=/  . .|     (/|     |. .  |=|  . .|
|    [UuU]    |. .  |/  .  .|     (v)     |. .  \=/  . .|     \=/     |. .  (v)  . .|      \|     |.   [UuU] .  | 8
|             |. . . . . . .|      ,      |. .  ( )  . .|     ( )     |. .   , . . .|             |. . . . . . .|
|             |. . . . . . .|             |. .  -.-  . .|      +      |. . . . . . .|             |. . . . . . .|
-----------------------------------------------------------------------------------------------------------------
|.   {===} .  |    {===}    |.   {===} .  |    {===}    |.   {===} .  |    {===}    |.   {===} .  |    {===}    |
|. .  (=)  . .|     (=)     |. .  (=)  . .|     (=)     |. .  (=)  . .|     (=)     |. .  (=)  . .|     (=)     |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             | 7
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
-----------------------------------------------------------------------------------------------------------------
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .| 6
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
-----------------------------------------------------------------------------------------------------------------
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             | 5
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
-----------------------------------------------------------------------------------------------------------------
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .| 4
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
-----------------------------------------------------------------------------------------------------------------
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|      /T     |. . . . . . .|             | 3
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|     (/|     |. . . . . . .|             |
|. . . . . . .|             |. . . . . . .|             |. . . . . . .|    {___}    |. . . . . . .|             |
-----------------------------------------------------------------------------------------------------------------
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|
|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .|             |. . . . . . .| 2
|     ( )     |. .  ( )  . .|     ( )     |. .  ( )  . .|     ( )     |. .  ( )  . .|     ( )     |. .  ( )  . .|
|    {___}    |.   {___} .  |    {___}    |.   {___} .  |    {___}    |.   {___} .  |    {___}    |.   {___} .  |
-----------------------------------------------------------------------------------------------------------------
|. . . . . . .|             |. . . . . . .|     -.-     |. .   + . . .|             |. . . . . . .|             |
|. . . . . . .|             |. .   , . . .|     ( )     |. .  ( )  . .|      ,      |. . . . . . .|             |
|.   [UuU] .  |      T\     |. .  (^)  . .|     / \     |. .  / \  . .|     (^)     |. . . . . . .|    [UuU]    | 1
|. .  | |  . .|      |\)    |. .  / \  . .|     | |     |. .  | |  . .|     / \     |. . . . . . .|     | |     |
|.   {___} .  |    {___}    |.   {___} .  |    {___}    |.   {___} .  |    {___}    |. . . . . . .|    {___}    |
-----------------------------------------------------------------------------------------------------------------
       A             B             C             D             E             F             G             H
```

Page last modified on October 04, 2008, at 02:32 PM

View Edit History Print

Search

simplaPoweredBy