Cigol-alpha-3.

Documentation. Released October 9, 2003

Cigol is open source software.

Contact Information:

For updates and information, go to: `cigol.sourceforge.net`.
Send queries or bug reports to: `songofwhispers@earthlink.net`.

If you have an interest in Cigol, please report any bugs, documentation errors, or design issues to the email address above.

This is the alpha 3 release of cigol.

To compile cigol, cd to the root cigol directory and run:

```
javac cigol/Cigol.java
```

Ziglar_Applet:

The Ziglar_Applet is an applet to run Cigol inside a web browser. It exists mainly to satisfy the odd hacker's curiosity. If you want to do serious work with Cigol, download the source code to your hard drive and run it on the command line.

The applet is available at cigol.sourceforge.net.

Changes since alpha 2.

- implemented a new mechanism to distinguish tokens that may have side effects from those that can't. This should allow more robust rule analysis, and make it easier to introduce new cigol commands. The old implementation of this functionality was an immature design.

- fixed situation with `every:` and `any:` evaluation which could sometimes return a value that, although correct, was not properly simplified.

- added conditional `if: then:` flow control.

- tweaked the thing that truncates long outputs to the command line interface. More information will be printed in full.

- added looping `for: do:` flow control. The `do:` half of the statement is allowed to have side effects, while the for: half of the statement acts like the `when:` statement in `explore-rule:`. This is unlike `explore-rule:` because explore-rule: is never allowed to have side effects.

- "improved correctness" with `explore-rule:`.

- fixed a number of stupid bugs with `representation()` methods.

- you can now assert lists. This asserts every element in the list. This mechanism also works to assert `explore-rule:`, which is equivalent to `assert: every: explore-rule:`, but executes in less time.

- you can now create and access external **contexts**; these are separate knowledge bases attached to each other. The pertinent Cigol directives are: `create-context:`, `context:`, `release-context:`, `name-context:`, `get-contexts:` . This implementation is currently immature.

- The statement of the form `relation: x y z` and the statement of the form `[x y z]` now have distinct meanings. The bracketed form evaluates to `true`, `false`, `unknown`, or a simplified form of itself. The `relation:` form always evaluates to a simplified form of itself without evaluating to `true`, `false`, or `unknown`.

- `quit:` is now an external function. This shouldn't change the correctness of any existing Cigol code, but it does mean that embedded Cigol scripts won't, by default, be able to kill their parent programs using `quit:`.

- `write:` allows you to write to stdout or stderr or to a file.

- Deleted macro functionality from the language entirely. (Text replacement macros are the scourge of the devil anyway.) Cigol now allows you to store rules at run time.

- Fixed bug where `symbol:` would allow you to declare a symbol whose representation is a restricted keyword. (i.e. `symbol: any`)

- Implemented `def-rule:`. This allows you to store a rule for later execution.

- Implemented `with: rule:`. Use

```
with:
  [variable-name-1 definition-1]
  [variable-name-2 definition-2]
  [variable-name-n definition-n]
rule: my-rule
```

You can use either a rule you stored using def-rule:, or enter a rule in-line. If there is only a single token after rule:, cigol assumes that token is the name of a rule defined

using def-rule.

- Implemented `new` to create new symbols. The new symbol is the concatenation of: an instance-global incrementing counter index, the time index of the moment cigol was launched, your user name, and your IP address. This makes it highly unlikely that any two calls to new will ever generate the same symbol.

- Implemented == for symbol identity. This is essentially a string comparison. So, `1` is = to `1.0`, but `1` is not == to `1.0`. Also, == can be used with any string, not just numerical values.

- Implemented a shortcut expression for certain `and` statements. This allows you to combine a large number of `and`-ed relations with the same primary symbol. For example: `[a [b c] [d e] [f g] [h i]]` is the same as `and [a b c] and [a d e] and [a f g] [a hi]`.

- It is now required that all relations be either bracketed or preceded with the `relation:` token. The unbracketed form was ambiguous in too many cases.

- Implemented arithmetic expressions. Expressions are in prefix notation, and the supported operators are: +, -, *, /, ^.

Basic Tutorial:

Launch Cigol using the java command:

```
your_shell% java cigol.Cigol
```

Let's imagine a simple logical system containing Norbert, Professor Smith, and Robotron. Norbert is a computer engineer working on Robotron, a robot who wants to become human. Norbert, being an engineer, is only happy when he's working on something interesting. Professor Smith, Norbert's engineering professor, is only happy when Norbert is working on his homework. Robotron is interesting, but Norbert's homework is not. Let's enter all of this information into Cigol. We use the `assert:` command to add new information to Cigol's knowledge base. Information is represented as a list of three symbols called a **relation**.

```
cigol(0)% assert: [norbert is-a computer-engineer]
assert: [norbert is-a computer-engineer]
null
cigol(1)% assert: [robotron is-a robot]
assert: [robotron is-a robot]
null
cigol(2)% assert: [prof-smith is-professor-of norbert]
assert: [prof-smith is-professor-of norbert]
null
```

```
cigol(3)% assert: not [norbert's-homework is interesting]
assert: not [norbert's-homework is interesting]
null
cigol(4)% assert: [robotron is interesting]
assert: [robotron is interesting]
null
cigol(5)% assert: [norbert is-working-on robotron]
assert: [norbert is-working-on robotron]
null
```

Now we've given Cigol some very basic information about our world. We can test any of the conditions that we just entered. Simply type the relation of interest to query Cigol about that relation:

```
cigol(6)% [prof-smith is-professor-of norbert]
[prof-smith is-professor-of norbert]
true
cigol(7)% [norbert's-homework is interesting]
[norbert's-homework is interesting]
false
```

Cigol answers that, yes, it is a true statement that Professor Smith is Norbert's professor. However, Cigol answers that Norbert's homework is not interesting. At this point, Cigol has merely regurgitated the same information that we he have already fed it.

Cigol is designed to draw deductive conclusions. What rules could we apply to cause Cigol to draw such conclusions? Well, for a start, Robotron is a robot. Therefore, Robotron is not human. That is, Robotron can not be both a human and a robot at the same time. He must be one or the other. But Cigol doesn't yet know this.

```
cigol(8)% assert: not and [robotron is-a human] [robotron is-a
robot]
assert: not and [robotron is-a human] [robotron is-a robot]
null
cigol(9)% [robotron is-a human]
[robotron is-a human]
false
```

Using our rule, Cigol has figured out that robotron is not human.

Now, Robotron is happy if and only if Robotron is human. We will need two rules to show this: 1) robotron is happy if robotron is human and, 2) Robotron is not happy if Robotron is not human.

```
cigol(10)% assert: and (if [robotron is-a human] then [robotron
```

```
is happy]) (if not [robotron is-a human] then not [robotron is
happy])
assert: and or not [robotron is-a human] [robotron is happy] or
not not [robotron is-a human] not [robotron is happy]
null
cigol(11)% [robotron is happy]
[robotron is happy]
false
```

Unfortunately, robotron is unhappy. Let's pause for a moment and see everything that we have learned about Robotron:

```
cigol(12)% print-description: robotron
(print-description: robotron)
true    : [norbert is-working-on robotron]
true    : [robotron is interesting]
true    : [robotron is-a robot]
false   : [robotron is happy]
false   : [robotron is-a human]
null
```

We notice, however, that the rules we just entered were very specific: if we added another robot, say, Data from Star Trek, we would need to rewrite all of our rules to determine whether or not Data was happy. If we had an army of hundreds of robots, we would need to write hundreds of different rules, and this would get uninteresting very quickly.

Cigol uses the `explore-rule:` construct to analyze more complex rules. Within the `explore-rule:` construct, we can define rules to describe all robots, all computer engineers, and even everything in the universe that is happy.

```
cigol(13)% assert: every: explore-rule: [norbert is happy] when:
(and [norbert is-working-on any PROJECT] [that PROJECT is
interesting])
assert: (every: (explore-rule: [norbert is happy] when: and
[norbert is-working-on any PROJECT] [that PROJECT is
interesting]))
null
cigol(14)% [norbert is happy]
[norbert is happy]
true
```

The `explore-rule:` directive is very, very powerful, and should not be used carelessly. Subtle errors can creep into your knowledge base if you do not clearly define every element of your logical system. In some cases, explore-rule: will need a great deal of processor time to complete its analysis. You should treat Cigol code the

same way you would code from any other language: code a little, test a little.

NOTE: `any` and `that` have precise implications that may not be obvious. In general, `any` means that Cigol will go looking through the knowledge base for symbols that belong in place of the `any` statement. `that` means that Cigol will use the value from another `any` statement, `def-var:` statement, or a parent `with: rule:` statement that uses the same variable name. If in doubt, use `any` for any case in which Cigol will allow it. All symbols referenced using `that` must be defined, and the `that` reference must be within the scope of that definition.

Finally, relations that use `any` or `that` for all three components of a relation force Cigol to search through its entire knowledge base to run rule exploration. This is a source of inefficiency. This bug-ish issue may be corrected at a future date.

Sometimes Cigol doesn't have enough information to solve a problem. For example, I never told Cigol whether or not Norbert is working on his homework.

```
cigol(15)% [norbert is-working-on norbert's-homework]
[norbert is-working-on norbert's-homework]
[norbert is-working-on norbert's-homework]
cigol(16)% if [norbert is-working-on norbert's-homework] then
[prof-smith is happy]
or not [norbert is-working-on norbert's-homework] [prof-smith is
happy]
or not [norbert is-working-on norbert's-homework] [prof-smith is
happy]
```

As you can see, Cigol is powerless to figure anything out without your explicit definitions of everything in its knowledge base. If you query Cigol about one of these cases, it will simply feed the expression back to you. In some of these cases, Cigol may offer you a simplification of your statement that does not resolve entirely to a true, false, or unknown truth value.

The "family" example uses Cigol to build a family tree.

To execute the "family" example, cd to the root cigol directory and run:

```
java cigol.Cigol
cigol(0)% evaluate: load-file: examples/sample-family.cigol
```

Use the `print-description:` command to examine the results. For example:

```
cigol(1)% print-description: susan
```

Cigol Commands:

```
and <expression> <expression>
```

Evaluates to the result of the logical `and` of the truth values of the two expressions.

Note the following truth table:

```
and true true          true
and true false         false
and true unknown       unknown
and false true         false
and false false        false
and false unknown      false
and unknown true       unknown
and unknown false      false
and unknown unknown    unknown
```

Note that the result of an `and` operation is the "least true" of the two expressions, in the sense that `false` is "less true than" `unknown` and `unknown` is "less true than" `true`. Can be asserted.

`any <symbol>`

Used for rule exploration. <symbol> becomes a variable name that can be referenced elsewhere in the rule using a `that` or another `any` token.

`any: <list>`

Combines all of the elements of a list into a logical expression using `or`. If the list is empty, the result is `false`.

`assert: <expression>`

Asserts a Cigol expression. This implies to the knowledge base that the expression is true, and attempts to store the result of that implication into that knowledge base. Once an assertion is made, it can not be retracted except by deleting the knowledge base.

`def-rule: <rule name> <expression>`

Stores a rule under the specified name. The name must be a symbol, but the expression may be any valid cigol expression.

Use `rule` to retrieve variable values.

`def-var: <variable name> <variable value>`

Variable names and values must be symbols, or any expression that evaluates to a symbol. Expressions can not be stored as variables.

Use `that` to retrieve variable values.

`describe: <symbol>`

Answers a list of all relations containing the specified symbol.

`evaluate: <expression>`
Evaluates the result of the specified expression. This is useful in some rare cases where the result of an expression is itself a valid unsimplified expression.

`every: <list>`
Combines all of the elements of a list into a logical expression using `and`. If the list is empty, the result is `true`.

`explore-rule: <rule> when: <when-rule>`
Answers conclusions from a rule. `any` and `that` can only be used within the context of rule exploration. The conclusions are returned as a list, as though you declared them explicitly using `list:`. Use `any:` or `every:` to combine the list elements into a logical expression.

`false`
Represents the truth value `false`. Can be asserted, but since assert: implicitly asserts that a statement is true, asserting false is always a contradiction, and Cigol will report a contradiction. `assert: not false`, on the other hand, is just fine.

`list: <list-element-1> <list-element-2> <list-element-3> . . .`
Defines a list of elements. Can be asserted.

`not <expression>`
Evaluates to the logical opposite of the truth value of the expression.

Note that it is often possible to build efficient systems without ever using `not`. Can be asserted.

`or <expression> <expression>`
Evaluates to the result of the logical or of the truth values of the two expressions.

Note the following truth table:

```
or true true          true
or true false         true
or true unknown       true
or false true         true
or false false        false
or false unknown      unknown
or unknown true       true
or unknown false      unknown
or unknown unknown    unknown
```

Note that the result of an or operation is the "most true" of the two

expressions, in the sense that `true` is "more true than" `unknown` and `unknown` is "more true than" `false`. Can be asserted.

`relation: <symbol> <symbol> <symbol>`
Relations consist of a list of three symbols, although complex relations may also contain `any` and `that` tokens. Relations may be (and are most often) encoded using '`[`'s and '`]`'s. `[<symbol> <symbol> <symbol>]` The form of relation preceded by the `relation:` token always evaluates to a relation, the form that is bracketed evaluates, if possible, to `true`, `false`, or `unknown`. Can be asserted.

`relation: (constructed form)`
To make is easier to manipulate complex data objects, it is possible to use a special construction. This takes the syntax:

```
[a
  [b c]
  [d e]
  [f g]]
```

This syntax is equivalent to `and and [a b c] [a d e] [a f g]`, and may include any number of paired elements.

`rule: <rule>`
Answers a rule stored with `def-rule:`. Can be asserted.

`symbol: <symbol>`
Symbols may contain any character except the left and right parenthesis, the left and right bracket ('{' and '}') and may not contain embedded quotation marks or whitespace. A symbol is the most basic unit of information in Cigol. The result of `symbol:` is simply the symbol that follows and is included only for completeness. A symbol enclosed in quotation marks may contain whitespace and is distinct from the same symbol no enclosed in quotation marks. (i.e. `foo` does not `== "foo"`.)

`that <symbol>`
Used for rule exploration. <symbol> must be a variable declared using an `any` token elsewhere in the same rule.

`true`
Represents the truth value `true`. Can be asserted.

`try:`
Normally, executing a logically contradictory assertion will invalidate the entire universe. Rules executed inside a `try:` can not invalidate the universe.

`unknown`
Represents the truth value `unknown`. Can be asserted, but Cigol will print a warning.

`while-learning: <expression>`
Repeats the evaluation of a Cigol expression for as long as new information is being added to the knowledge base.

```
with:
  [<variable name 1> <variable value 1>]
  [<variable name 2> <variable value 2>]
  . . .
  rule:
    <expression or stored rule name>
```
Evaluates a rule stored using `def-rule:` or a rule expressed in-line using the specified list of variable assignments. Can be asserted.

External Commands:

These external commands are available from the command line only and are stored in cigol.externals.ExternalFunctionLibrary. If you use the programmatic interface, you will need to load these external functions manually.

`load-file: "foo.cigol"`
Loads and returns the contents of a valid Cigol source file. Use `evaluate: load-file:` to execute a file. Place file names in quotation marks if they contain white space or special symbols. File names may not themselves contain quotation marks.

`print-description: foo`
Replacement for the old `describe:` function. Prettier.

`create-context: foo`
Creates a new external knowledge base ("context") named foo-context.

`context: foo (assert: [a b c])`
Accesses the external context named foo and evaluates an expression within that context. Answers the result of evaluating that context.

`release-context: foo`
Releases the external context named foo and all memory used by that context. Essentially destroys all data inside foo. If foo is stored under more than one name, it must be released for each name before the data therein will be forgotten. You can not release the root context, although you can make it unreachable.

`get-contexts:`
Answers a list of all contexts associated with this context.

`name-context: foo`
Names the current context. A context may have more than one name. This is mainly used to name the root context which, be default, has no assigned name.

```
write: stdout "This is a simple message to the world!"
write: stderr "This is an error message."
write: hello.txt "This message was sent to a file."
write: /dev/null
    "This message is headed for the bit bucket.  Bye. (*NIX only)"
write: saved-data.cigol
    (for: [any a any b any c] do: relation: that a that b that c)
```

Sends data to one of the two output streams or to a file.

Change History Before alpha 2: Changes since alpha 1.

- `do:` has been changed to `list:`. The new list mechanism is intended to encode a list of any type of element, not just directives. `explore-rule:` now evaluates to a list.

- Use `every:` to combine all of the elements of a list into one massive and-ing of the list. Use `any:` to do the same thing as one massive or-ing. You can do this with any list, not just the lists returned from `explore-rule:`.

- `explore-rule:` has been changed to an `explore-rule: when:` construct. Within this construct, the rule will be analyzed only when the result of the `when:` clause is `true`. For example:

```
  explore-rule: if (and
                      [that HACKER is-working-on that PROJECT]
                      [that PROJECT is interesting])
                  then [that HACKER is happy]
    when: (and:
            [any HACKER is-a hacker]
            [any PROJECT is-a project])
```

This new construct can be used to restrict rule exploration to appropriate situations. Used correctly, it may speed up your queries. Used incorrectly, it may prevent Cigol from deriving necessary rules.

In the above example, if the `HACKER` is not a `hacker`, or the `PROJECT` is not a `project`, then Cigol will not evaluate the rule for that `HACKER` or that `PROJECT`. For this to work, you must assert anything that is-a hacker or is-a project, otherwise the rule exploration will yield an empty list.

- Cigol now truncates extremely long command lines. Look for `. . .` `<LINE CONTINUES>` This is especially useful when executing files.

- Cigol now recognizes comments. Any text placed between the `{` and `}` characters will be ignored.

- Cigol now recognizes quoted strings. The token generated by a quoted string will normally include the quotation marks themselves. Thus, `is-a` is not the same as `"is-a"`. Quoted strings may contain any character other than the quotation mark itself. Cigol will never be modified to interpret escape sequences in strings.

- Fixed bug with `list:` representation() method.

- When parsing lists, Cigol now assumes the default list element type is a symbol rather than a relation. Explicitly use brackets or parenthesis to denote relations. This only matters when no effort is made to denote the list element type.

- Fixed issue with `def-macro:` where macros defined within a file might not be usable within that file. Macros are now defined both when the `def-macro:` directive is parsed and again whenever the `def-macro:` directive is evaluated. As a result, def-macro is BOTH a preprocessor directive AND a runtime command. Note, however, that when used as a runtime command, any expressions that have already been expanded with that macro will not be updated. The user should realize that redefining macros can have unexpected side effects (which has been true with all text replacement based macros since the dawn of the computer).

- Constructed the programmatic interface, represented by the cigol.CigolContext class. Instantiate a CigolContext to control Cigol from within another java program.

- Implemented external java language hooks. External hooks are called just like regular cigol directives.

- Added the first external function. `load-file:` loads and returns a file. Use `evaluate: load-file:` to execute the file.

- `describe:` now returns a list instead of printing data to the screen. (Remember, in text UI mode the data will still print to the screen.) This is useful when using the programmatic interface, and may have some other uses.

- if you were using `describe:` to print text descriptions of certain entities, try using the new improved `print-description:` instead. `print-description:` sorts elements into true and false as well as alphabetically.

- The definition of `evaluate:` has changed. It now evaluates the result of evaluating the expression that follows it. The old version of `evaluate:` really didn't change the expression. At all.

- The definition of `try:` has changed very slightly. This is not likely to affect anything.

The issue where `while-learning:` would infinite loop in a `try:` has been fixed.

- Added `def-var:` to define variables.

- Various other minor changes.