The Giles Production Rule System Compiler

User Manual

The KoreLogic Development Team

1 Introduction

Giles [1] is a compiler that creates *production systems* (or "engines" in Giles parlance).

A production system is a program that is typically used to provide some sort of artificial intelligence. Giles engines are particularly well-suited to being expert systems, log analyzers, and behavior-detection systems.

Giles is a compiler, meaning that it is not itself a production system, but rather a tool for creating production systems. In fact, Giles's defining feature is that its output is not a program in the traditional sense, but rather a schema for a modern, unmodified, SQL-based Relational Database Management System (RDBMS). That means that Giles can turn a normal, unmodified RDBMS into a fully functional production system.

This approach has immediate advantages. Perhaps the most important is that programmers can use normal database access interfaces, which are ubiquitous; this immediately makes Giles engines accessible to far more programmers than those of traditional systems. The engines are also able to take advantage of modern databases' data safety and reliability guarantees, as well as their transactional semantics. This means that Giles engines are capable of dealing with huge amounts of data safely, even in the face of system crashes, and over long periods of time. These benefits are important given the expected use cases of Giles engines: pre-defined production systems that are built into and provide intelligence for larger, potentially long-running and unattended, systems.

In other words, Giles lets you deploy a production rule system anywhere where you can deploy your (supported) database.

This document describes Giles-the-compiler and Giles-the-language. It assumes you (the reader) have a decent understanding of SQL, but does not require that you have any experience with production systems or artificial intelligence. In fact, one of Giles's goals is to make production systems (often seen as esoteric) more accessible.

[1] Giles is named after the character Rupert Giles from the *Buffy the Vampire Slayer* television and comic book series. The names are pronounced identically, with a soft *G*.

1.1 What is a Production System, Anyway?

A production system (or *production rule system*) is a kind of computer program that is often used to provide some sort of artificial intelligence. Production systems are very often used for log and behavior analysis and expert systems. Production systems consist of *facts* and *rules*.

A fact is a single datum describing a discrete piece of information in the problem domain of the system. Some example facts might be:

- Suzanne is an astronaut.
- Train #447 arrived at platform $9\frac{3}{4}$ at 08:00.
- User "jdoe" logged in via terminal #16 at 12:33.

The set of all known facts is referred to as the *working memory* of the system.

A rule is a simple "if-then" statement. The "if" part is known as the *predicate*, and specifies a pattern of facts. The "then" part is known as the *action*. When some set of facts in working memory matches the predicate, the action is performed.

Giles defines two possible actions: *assert* and *suppress*. The assert action adds a new fact to the working memory, while the suppress action suppresses facts already in working memory. When a rule's action is taken, it is said that the rule has *fired*. When a rule fires, it can add or remove facts from working memory, which can cause other predicates to match, which can cause other rules to fire, and so on.

Rules only work in the presence of facts. They aren't like more traditional computer programs that have well-defined start and end points. Instead, a production system sits in a cycle that looks like this:

Step 1 Find all rules whose predicates are matched by a set of facts in working memory. If there aren't any matching predicates, halt.

Step 2 Perform the actions for those rules.

Step 3 Go to Step 1.

This is called the *recognize-act* cycle, and it executes once for each fact that is added to or removed from working memory.

In other words, when a production system starts up, it just sits there, waiting for the user to add some facts to working memory. As facts are added and removed, the production system updates the contents of working memory by adding and removing other facts.

The general workflow for a production system is therefore to add or remove some facts, and then examine working memory to see what it looks like after those changes. For example, a log analyzer might have facts in memory representing individual log entries, and other facts representing "alerts" that indicate something supsicious was inferred from those logs. The user would add one fact per log entry, and then check to see if any alert facts are present.

While this all might sound fairly simple, the trick is doing this all efficiently. A given engine might have hundreds of rules and millions of facts, so finding matching predicates quickly is critical to performance. That's where Giles comes into play: it creates efficient production systems[2].

1.2 What Are Giles-Style Production Systems Good For?

Giles creates production systems that are good for finding patterns in large data sets where data is added or removed incrementally over time. Some example use cases:

Log Analysis For example, an engine could be given one fact per log entry, and could correlate multiple log sources to find complex behavioral patterns.

Inference Engines As facts are added to an engine, a more complete picture can be built up of a system. For example, a network scanning tool might assert facts about discovered devices and the engine can infer client-server relationships, device operating systems, and other pieces of complex information.

Expert Systems Facts can be asserted listing symptoms. The individual symptoms could be correlated to infer possible diagnoses. As more diagnoses are added from more and more patients over time, inferences can be made about epidemiological data.

1.3 What You Need to Already Know

Giles's most interesting feature is that it compiles a production system description in a SQL database schema. This manual assumes that you are already at least somewhat familiar with SQL. If not, there are many excellent tutorials available on the Web. The examples in this manual all use the ubiquitous and excellent SQLite database engine, and its dialect of SQL is documented at `https://www.sqlite.org/lang.html`.

Giles's input takes the form of YAML (`http://www.yaml.org`) files. YAML, which stands for "YAML Ain't Markup Language" is a human-readable data serialization language. Some familiarity with YAML might come in handy when reading this manual, but it is not strictly necessary. (A nice side effect of using YAML for Giles's input language is that there are many automated tools that can process YAML files, making it easy to write programs that can manipulate Giles source code.)

2 Getting Your Feet Wet

The best way to learn anything is by following examples. This section describes a basic Giles engine piece by piece, to give you a flavor of the language and the

[2]Well, as efficient as possible given the rules that were written. It's possible to write inefficient programs in any language.

output engine.

2.1 Facts, Fact Classes, and Rules

In Giles, facts are made up of fields. For example, a fact that describes activity in a train station might have fields for train number, arrival time, departure time, destination, etc. A fact that describes a person might have fields for first and last name, age, and home address.

In Giles, every fact belongs to exactly one *fact class*, and a fact's class determines what fields it has. Every engine has a section called *Facts* that describes the fact classes known to the engine.

Every fact has every field filled in — there is no concept of default or `NULL` values for fields.

In engines, facts are just rows in normal database tables. New facts are added to working memory ("asserted") by inserting rows into tables, and facts are removed from working memory ("retracted") by deleting rows. Giles guarantees (thanks to database transactions) that fact tables are always globally consistent; after a fact is inserted or removed, all of the fact tables in the system are updated atomically and can be inspected without fear of inaccurate or missing data.

Facts asserted by the user are always distinct. That is, the user is free to assert as many facts of the same class and with identical fields as desired. This is important in situations where multiple facts may be intrinsicly indistinguishable (say, for example, in an aritificial intelligence problem dealing with differently colored blocks, some duplicated, where any red block is sufficient). The user can always add a "distinguishing" field to a fact class (a name, for example) if necessary. Automated processes can deduplicate data as required.

When Giles compiles an engine, it turns rules into triggers on fact tables. That way, when a new fact (row) is inserted or removed from working memory (tables), these triggers fire and perform all of the actions defined by the rules.

That means that working with a Giles engine is as simple as performing normal SQL `INSERT`, `DELETE`, and `SELECT` statements. What could be easier?

2.2 All Men Are Socrates

Without further ado, here is a simple engine that infers that if someone is human, he or she is mortal:

```
1  Facts:
2      IsHuman:
3          Person: STRING
4
5      IsMortal:
6          Person: STRING
7
8  Rules:
9      AllHumansAreMortal:
10         Description: All humans are mortal.
```

```
11
12          MatchAll:
13              - Fact:    IsHuman
14                Meaning: The named person is human.
15                Assign:
16                  Person: !expr This.Person
17
18          Assert:
19              IsMortal:
20                  Person: !expr Locals.Person
```

Engine descriptions are purely declarative; it helps to keep this in mind while reading through this section. Let's look at this engine piece-by-piece:

Lines 1–6

Here we define two fact classes, `IsHuman` and `IsMortal`. `IsHuman` facts are assertions that some person is human, while `IsMortal` facts are assertions that some person is mortal.

Lines 8–10

Here we define the rules for the engine; this engine only has one. This rule is named `AllHumansAreMortal`. We give the rule a human-readable description (such a description is mandatory for reasons that will be explained later).

Lines 12–16

Here is the predicate for the rule. All predciates have at least a `MatchAll` clause, which states "this clause is true if all of my sub-clauses are true". In this case, there is just one sub-clause, introduced by the `-` (hyphen).

The sub-clause declares a fact class to test for (`IsHuman`), and gives a human-readable description of what that fact means.

The sub-clause also performs an *assignment*. Assignments extract parts of matched facts and store them in *local variables*. These local variables can then be used in expressions in other parts of the predicate; it is through local variables that facts can be interrelated by rules.

Remember that engine descriptions are declarative. While "assignment" might sound procedural[3], it's still declarative. Predicates are completely order-insensitive, meaning that the facts that they match can be inserted into working memory in any order. In light of this, it might have been better to call it "extraction" rather than "assignment".

The assignment here creates a new local variable called `Person` and gives it a computed value based on an expression. The `!expr` tag indicates that a dynamic expression is to follow. These expressions can be arbitrarily complicated, but in this case, the expression is simple: it's just the value of the `Person` field of

[3] Okay, it *does* sound procedural. Sorry.

the fact matched by this clause (the `This` specifier refers to "the fact matched by this clause").

A new "instance" of a rule is created for each matching set of facts in working memory. Since there is only one matching clause in our predicate, that means there will be a new instance of this rule for each `IsHuman` fact in working memory, and each instance will have its own value for the `Person` local variable.

Lines 18–20

Here we define the rule's action. In this case, it's `Assert`, meaning we're going to add a new fact to working memory. Assert actions have a single sub-clause, which is the name of a fact class (in this case, `IsMortal`), which itself contains the values that populate the fields of this new fact.

On line 20, we provide a value for the new fact's only field (`Person`). Note that this is a dynamic expression, as indicated by the `!expr` tag. Again, this expression could be arbitrarily complicated, but we go for simple in this example: just the value of the `Person` local variable.

This assertion completes our rule: we infer that if someone is human, he or she is mortal.

2.3 Compiling and Loading

Let's compile this engine. Put the engine description above into a file called `mortal.yml` and run this command:

```
# giles -o mortal.sql mortal.yml
```

This will, assuming nothing goes wrong, give us a SQLite database schema in `mortal.sql`.

Let's create a database by loading this schema into a (completely normal!) SQLite instance:

```
sqlite> .read mortal.sql
```

We now have a working production system.

2.4 Asserting and Viewing

Let's assert a new fact, asserting that Socrates is human:

```
sqlite> INSERT INTO Giles_IsHuman_Facts(Person)
   ...> VALUES('Socrates');
```

Facts that are asserted like this (that is, directly by the user) are referred to as *axioms* or *axiomatic facts*. That is in contrast to facts that are asserted by the engine, which are *derived facts*.

Now that we've made this assertion, let's see if our rule worked and asserted that Socrates is mortal:

```
sqlite> SELECT * FROM Giles_IsMortal_Facts;
Person = Socrates
Id     = 1
```

Success!

The `id` field stores a unique identifier that is automatically added to every fact. The `id` is guaranteed to be unique within a given fact class, and so a fact class plus an `id` is enough to uniquely identify any fact.

2.5 Justifying

Now that we've inferred that Socrates is mortal, we can ask the engine to *justify* its knowledge:

```
sqlite> SELECT Justification
   ...> FROM Giles_IsMortal_Justification
   ...> WHERE Id = 1;
Justification =
    Fact 'IsMortal #1' was produced
    by rule 'AllHumansAreMortal':

    All humans are mortal.

    Justification:
        * The named person is human. (IsHuman #1)
```

The engine has justified how it knows that Socrates is mortal — namely, via the rule `AllHumansAreMortal` and because he `IsHuman`. This justification could be applied recursively, all the way back to the axioms that originally lead to this conclusion.

2.6 Retracting

We can now retract our initial assertion that Socrates is human:

```
sqlite> DELETE FROM Giles_IsHuman_Facts
   ...> WHERE Person = 'Socrates';
```

With its support gone, the inference that Socrates is mortal is now gone too:

```
sqlite> SELECT COUNT(*) FROM Giles_IsMortal_Facts;
COUNT(*) = 0
```

2.7 The Fundamental Giles Guarantee

Why did the inference that Socrates is mortal vanish? Because of the fundamental guarantee of all Giles engines:

- All facts that can be derived from the current set of axioms in working memory are derived.
- No facts that cannot be derived from the current set of axioms in working memory are derived.

That is to say, when the user inserts axioms into working memory, or retracts them from working memory, the engine guarantees that all of working memory

is updated (atomically!) to reflect the set of derivable facts based on those axioms.

That's why, when we retracted our assertion that Socrates is human, the engine no longer provided the fact asserting he was mortal.

This guarantee is always true, and extends no matter how far cause is from effect. Even if 6.02×10^{23} rules had to fire to make the inference, the engine will remove the fact if it can no longer be supported by the set of axioms in the working memory.

This guarantee has some useful consequences. Users can "experiment" with adding and removing facts safe in the knowledge that nothing will be lost that can't be restored, and no unreliable data can be produced. For example, users could create an expert system that knows about network design, and experiment with different network layouts by simply adding and removing facts, to see what the expert system thought about the network design.

3 Fact Classes

As was stated earlier, all facts are instances of exactly one fact class. Engines must define at least one class of facts.

Fact classes simply specify a name and the names and types of the class's fields. A class must have at least one field.

Fact classes are defined in the `Facts` section of an engine. For example:

```
1  Facts:
2      Person:
3          FirstName:  STRING
4          Age:        INTEGER
5      Car:
6          Year:       INTEGER
7          Make:       STRING
```

3.1 Field Types

Every field is of some type. Four types are defined:

BOOLEAN Either `TRUE` or `FALSE`.

INTEGER An integer of some width.

REAL A floating point number of some precision.

STRING A string of characters in some encoding.

The widths, precisions, and encodings of these types are intentionally vague, because they depend on the target database. In general, integers are at least 32 bits wide, but might be 64 bits wide or even of infinite width. Real numbers are usually IEEE double-precision floating point numbers, but might be single precision.

String encoding is the biggest variable, though. Some systems use ASCII, some use EBCDIC, some support UTF-8, some don't.

Consult your database's documentation to get exact limits and semantics for these types.

3.1.1 Type Checking

Giles performs full and strict type checking on engines during compilation. Type transfer functions are available to cast values of one type to another.

The following functions are available:

string_of_bool Cast a boolean to a string value.

string_of_int Cast an integer to a string value.

string_of_real Cast a real to a string value.

int_of_real Cast a real to an integer value.

int_of_string Cast a string to an integer value.

real_of_int Cast a integer value to a real value.

The exact semantics of these functions varies depending on the targeted database system. There is a general guarantee, however: none of these functions will ever result in a `NULL` value or abort the current transaction. Failed conversion is always signaled via some database-specific canary value.

One important note about the strict type checking is that there is no silent promotion of types. It is not possible, for example, to freely mix `REAL` and `INTEGER` values in arithmetic expressions; explicit type transfers must be performed.

3.2 !output Facts

Giles performs a special optimization called α-pruning. This optimization essentially ignores facts that can be cheaply proven to never match any predicate defined in the engine. "Ignored" in this case means "completely ignored", in that inserting such a fact into a table has no effect — not even inserting a row into the table.

This is an important optimization, because it means that facts that can never have an impact on the engine take up no storage space. It does however, come with a price: the user might be interested in some facts even if the engine never is (say, "alerts" or something). Giles allows fact classes to be spared from α-pruning by marking the class as `!output`.

For example:

```
1 Facts:
2     AnOutputFact: !output
3         Message: STRING
```

Giles automatically marks fact classes that only appear in actions as `!output`. That is, if a fact class never appears in a predicate, it is always an `!output` class. This automatic marking handles the vast majority of cases where we have seen facts marked as `!output` in practice.

4 Rules

Every rule consists of exactly one predicate and exactly one action.

Because Giles's expression language is used extensively throughout the various parts of rules, it might be useful to make a small diversion now to discuss the expression language in more detail.

4.1 Expressions and the Basic Operators

Remember that Giles's input language is just YAML. The compiler makes good use of YAML's native type scheme, meaning literal numbers and strings and such can be written "naturally" and things will generally work out fine. However, this can only express static values, and there are situations where we need dynamic (that is, calculated at runtime) values.

Enter the `!expr` tag. A string tagged with `!expr` is passed to Giles's expression parser instead of being treated as a normal string.

Giles's expression parser is pretty straightforward: it uses normal infix notation, with more-or-less normal precedence, and with parentheses to change precedence as needed. Functions can be invoked in the "standard" name-and-parenthesized-argument-list notation.

Here are some sample Giles expressions:

```
!expr 1 + 1
!expr strlen("hello, world!")
!expr (Locals.Foo + 7) >= 30 &&
    strlen(This.Name) <= 39
```

The basic binary operators are listed here, in order of precedence:

Operator	Input Type	Result Type	Description
`*`	*numeric*	*numeric*	Multiplication
`/`	*numeric*	*numeric*	Division
`%`	`INTEGER`	`INTEGER`	Remainder
`+`	*numeric*	*numeric*	Addition
`-`	*numeric*	*numeric*	Subtraction
`.`	`STRING`	`STRING`	Concatenation
`==`	*any*	`BOOLEAN`	Equality
`!=`	*any*	`BOOLEAN`	Not Equal
`<`	*numeric*	`BOOLEAN`	Less Than
`>`	*numeric*	`BOOLEAN`	Greater Than
`<=`	*numeric*	`BOOLEAN`	Less Than or Equal
`>=`	*numeric*	`BOOLEAN`	Greater Than or Equal
`~`	`STRING`	`BOOLEAN`	Regex Match (see 4.1.3)
`&&`	`BOOLEAN`	`BOOLEAN`	Conjunction
`||`	`BOOLEAN`	`BOOLEAN`	Disjunction
`AND`	`BOOLEAN`	`BOOLEAN`	Predicate Join (see 4.2)

There are also two unary operators:

Operator	Input Type	Result Type	Description
`-`	*numeric*	*numeric*	Arithmetic Negation
`NOT`	`BOOLEAN`	`BOOLEAN`	Logical Negation

Unary operators have precedence over all binary operators.

4.1.1 YAML Quirks

Sometimes expressions need to be parenthesized in a certain way to work around YAML parsing syntax. For example, this expression is not valid YAML:

```
!expr "Hello, " . "World!"
```

because it starts with a double quote, and the YAML parser therefore only parses the string up to the next double quote.

To get around this, parentheses may be used. The above expression could be safely rewritten like this:

```
!expr ( "Hello, " . "World!")
```

Note the extra space after the first parenthesis, which is required to ensure the YAML parser doesn't treat the expression as a simple string.

4.1.2 Constants, Field References, and Local Variables

There are three namespaces that are visible in expressions. All variable (and named constant) references are qualified with their namespace.

Named constants are placed in the `Constants` namespace. Constants are defined in the `Constants` section of the engine. For example:

```
Constants:
    Pi: 3.14
    TheAnswer: 42
```

These constants could be referred to as `Constants.Pi` and `Constants.TheAnswer` in expressions. The type of a constant is automatically inferred from its declaration.

Field references are valid only in expressions in sub-clauses of predicate clauses. The fields of the fact matched by that sub-clause are placed into the `This` namespace. For example:

```
    MatchAll:
        - Fact:    Person
          Meaning: A person named Douglas who is
                   at least 42 exists.
          When:    !expr This.FirstName == "Douglas" AND
                         This.Age >= 42
```

Field references can be used inside assignments as well, to extract values from facts and into local variables.

Local variables are created by assignments in `MatchAll` predicate clauses. The engine automatically creates on "instance" of a rule per matching set of facts in working memory and each instance has its own set of local variables. Local variables are placed in the `Locals` namespace. For example:

```
    MatchAll:
        - Fact:    Person
          Meaning: A person exists.
          Assign:
            FirstName:  !expr This.FirstName

        - Fact:    Dog
          Meaning: A dog exists owned by that person.
          When:    !expr This.Owner == Locals.FirstName
```

Local variables are the mechanism by which multiple facts can be related to one another. The assignment of a local variable can use an arbitarily complex value (that is, the assignment is an expression).

Note that, for technical reasons, the assignment of a local variable must occur textually prior to its first use. This affects how sub-clauses are ordered, but should not in any way be taken to mean that the matched facts must be asserted in some order; predicates are always order-insensitive.

4.1.3 Regular Expressions

Note that the `~` operator allows regular expressions to be used in engines. However, regular expressions are not supported by all targeted database backends, and on others may require special configuration[4].

[4]This is the case in SQLite.

To make sure the user is prepared to configure the target database appropriately, the compiler will fail to compile an engine using regular expressions unless the compiler is invoked with the `-r` flag.

4.2 Predicates

Every rule has a predicate that specifies a pattern of fact(s) to match. Every predicate has at least a `MatchAll` clause, and, optionally `MatchNone` and `When` clauses. All of the clauses specified in a predicate must be true for the predicate to be true and the rule to fire.

Predicates are always order-insensitive: the facts matched by a predicate can be asserted in any order. Also, the same fact can match multiple different clauses in a predicate at once. Finally, the engine will find *all* sets of facts that match a predicate and fire the rule once for each set.

4.2.1 MatchAll

The `MatchAll` clause specifies a set of sub-clauses, each matching a single fact. *All* sub-clauses of a `MatchAll` clause must be true for the clause itself to match.

A `MatchAll` clause looks like this:

```
1  MatchAll:
2      - Fact:    Person
3        Meaning: A person named Socrates exists.
4        When:    !expr This.FirstName == "Socrates"
5        Assign:
6          FirstName: !expr This.FirstName
7
8      - Fact:    Citizenship
9        Meaning: That person is Athenian.
10       When:    !expr This.FirstName == Locals.FirstName AND
11                     This.Polis == "Athens"
```

Each sub-clause begins with a hyphen. This is a YAML-ism to indicate that the clauses are a list.

Each sub-clause identifies a fact class, and provides a human-readable meaning to be used during justification. Each sub-clause may also optionally have a complex predicate (the `When`) portion, and a set of assignments.

If a sub-clause has a complex predicate, it must be a list of *field reference-comparator-value* triples conjoined with the special `AND` operator. The `AND` operator acts like a normal boolean conjunction, but combines tests of field values together into a complex predicate.

If a sub-clause does not have a complex predicate, then the clause will match *any* fact of the appropriate class in working memory (that is, there is a default "true" predicate).

4.2.2 MatchNone

The `MatchNone` clause specifies a set of sub-clauses, each matching a single fact. If *none* of the sub-clauses of a `MatchNone` clause match, then the clause itself matches.

Each sub-clause of a `MatchNone` clause is of the same form as the `MatchAll` sub-clauses, except that assignments are not allowed.

4.2.3 Final When

A rule may have a final predicate, known as a `When` predicate. It is simply a single expression of `BOOLEAN` type that must be true for the rule to fire. For example:

```
1  When: !expr Locals.Foo == TRUE AND Locals.Bar == 39
```

4.3 Actions

Every rule has exactly one action. Two types of actions are possible: `Assert` (add a new fact to working memory) and `Suppress` (remove some facts from working memory). Each of these actions modifies working memory in some way, which can cause more rules' predicates to match, causing those rules to fire, and so on.

4.3.1 Assert Actions

The `Assert` action asserts a new fact into working memory. The action simply takes the name of a fact class and a dictionary specifying the new fact's fields' values. For example:

```
1  Assert:
2      Car:
3          Make:  Yoyodyne
4          Model: Tristero
5          Year:  !expr Locals.Year
```

4.3.2 Suppress

The `Supress` action retracts facts from working memory. It simply takes the name of a fact class and a predicate and retracts all facts that match.

For example:

```
1  Suppress:
2      Fact: Car
3      When: !expr This.Make == 'Yoyodyne'
```

Note that the Giles Guarantee (see 2.7) still holds in light of `Suppress` actions. For example, say rule `R` matches fact `F` and fires, and as its action suppresses fact `F2`. If fact `F` is later suppressed by a rule or retracted by the user, fact `F2` will be restored. This ensures that all derivable facts are derived.

However, if the user manually deletes some axioms, things are really and truly gone. The engine guarantees that all facts derivable from a set of axioms are derived, but deleting an axiom changes that set, and thus changes the set of derivable facts.

5 Parameters

Parameters are a special kind of fact class, declared in a special `Parameters` section of an engine. These parameters can have limits placed on their values and can be guaranteed to be singletons, or multiply-valued and indexable by a string, and have default values. While they are just normal facts under the hood, they can provide a useful "tuning" interface.

Some example parameters:

```
1  Parameters:
2      ABooleanParameter:
3          Default: FALSE
4
5      AnIntegerParameter:
6          Default: 100
7          Lower:   0
8          Upper:   100
9
10     ARealParameter:
11         Default:  1.0
12         Lower:    0.0
13         Upper:    1.0
14
15     AStringParameter:
16         Default:  "Hello, world!"
17
18     ADictionaryParameter:
19         Dictionary: TRUE
20         Default:    FALSE
```

The engine provides a special interface to the parameters by way of a `Parameters` table. Modifying a parameter's value will fail if an invalid parameter name is specified, or if the provided value falls outside of the specified limits.

Parameters create fact classes of the same name with a single field of the appropriate type named `Value`. Dictionary parameters have an additional field of type `STRING` named `Key`. Parameters are matched in predicates just like any other fact.

6 Recursive Rules

The matching process in an engine is inherently recursive: when a rule's action modifies working memory, more rules may fire as a result. However, as long as

there are no cycles among those rules (that is, none of those rules asserts or suppresses facts that are matched by another of those rules), the processing is sure to terminate.

Problems arise when cycles like these exist in a rule set. Computation might go on forever in a recursive loop or the database could raise an exception when stack space is exhausted.

Giles supports rules with these sorts of cycles[5], with the caveat that undefined behavior can result if recursion is not terminated or continues on long enough to exhaust some finite resource provided by the hosting database.

To write these sorts of rule sets, Giles allows an `Assert` action to be decorated with a `!distinct` tag. This tag means that a new fact will be asserted if and only if an identical fact does not already exist. If such a fact already exists, the rule's action will not fire. This can be used to break recursive loops.

Note that for the purposes of the `!distinct` tag, "identical" means "of the same fact class and with identical values for all its fields".

Note that the Giles Guarantee (2.7) still applies in this case. If a fact that is produced distinctly is suppressed, the engine will check to see if any of the other rules that produce that fact could fire; if so, they do. Thus a suppressed fact is only suppressed if there are no rules that could produce the fact that are capable of firing.

Because an engine with cycles in its rule sets can have undefined behavior, the compiler will not compile such sets unless it is invoked with the `-c` flag, which indicates that the user is aware of the cycles and their potential consequences.

It is not possible to suppress a fact belonging to a class that appears in any `!distinct` assertion, because it could cause the engine to enter an infinite loop. Additionally, not all database backends will support these sorts of rule sets, and others may require special configuration.

7 The Foreign Function Interface

Any single-valued function provided by the host database can be used in an expression by declaring it via the foreign function interface. This declaration must state the number and types of arguments, the name of the function in the database, the name will function will be known as in Giles, and the return type.

For example, the SQLite `SUBSTR` function has a declaration similar to this:

```
1     Functions:
2         Substr:
3             External:   SUBSTR
4             Parameters: [STRING, INTEGER, INTEGER]
5             Returns:    STRING
```

This declares the Giles function `Substr` is implemented by the SQLite function `SUBSTR`. It takes three parameters: a `STRING` and two `INTEGER`s. It returns a value of type `STRING`.

[5] Assuming the underlying database engine supports recursive triggers; not all do, and some require special configuration.

With this declaration, the `Substr` function can be used in any Giles expression transparently.

Note that foreign functions must return a single value of the appropriate type, and must never return `NULL` (doing so results in undefined behavior).

8 Multi-File Engines

An engine definition can be split across multiple YAML files. Each file can have any combination of the top-level sections (for example, `Facts` or `Parameters`). All of these top-level sections will be merged together during compilation.

This is a useful structuring technique as engines grow larger.

9 Database-Specific Notes

This section outlines the some notes and caveats specific to different database backends supported by the compiler.

9.1 SQLite

- SQLite version 3.7.16.1 or greater is required.
- Recursive triggers must be enabled.
- Foreign key support must be enabled.
- By default, SQLite does not support regular expressions. If regular expressions are used in rules, the user must provide regular expression implementation that is compatible with the Perl Compatible Regular Expression (PCRE) library.
- SQLite has a default limit of 1000 recursive trigger invocations. An engine without cycles in its ruleset would be very unlikely to ever run into this limit, but with cycles this limit could be reached very quickly. Exceeding this limit results in a runtime error.