

1.4. Rete Algorithm

The RETE algorithm was invented by Dr. Charles Forgy and documented in his PhD thesis in 1978-79. A simplified version of the paper was published in 1982 (<http://citeseer.ist.psu.edu/context/505087/0>). The word RETE is latin for "net" meaning network. The RETE algorithm can be broken into 2 parts: rule compilation and runtime execution.

The compilation algorithm describes how the Rules in the Production Memory to generate an efficient discrimination network. In non-technical terms, a discrimination network is used to filter data. The idea is to filter data as it propagates through the network. At the top of the network the nodes would have many matches and as we go down the network, there would be fewer matches. At the very bottom of the network are the terminal nodes. In Dr. Forgy's 1982 paper, he described 4 basic nodes: root, 1-input, 2-input and terminal.

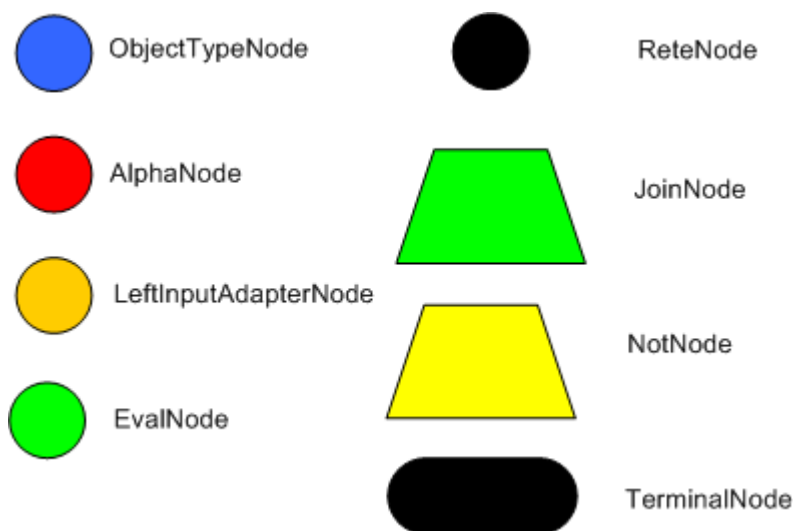


Figure 1.5. Rete Nodes

The root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeName. The purpose of the ObjectTypeName is to make sure the engine doesn't do more work than it needs to. For example, say we have 2 objects: Account and Order. If the rule engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the engine should only pass the object to the nodes that match the object type. The easiest way to do this is to create an ObjectTypeName and have all 1-input and 2-input nodes descend from it. This way, if an application asserts a new account, it won't propagate to the nodes for the Order object. In Drools when an object is asserted it retrieves a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class; if this list doesn't exist it scans all the ObjectTypde nodes finding valid matches which it caches in the list. This enables Drools to match against any Class type that matches with an instanceof check.

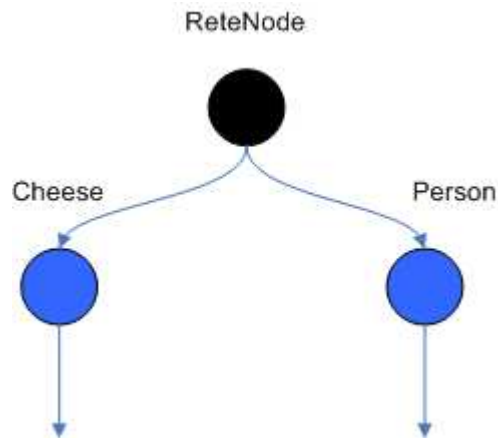


Figure 1.6. ObjectTypeNodes

ObjectTypdeNodes can propagate to AlphaNodes, LeftInputAdapterNodes and BetaNodes. AlphaNodes are used to evaluate literal conditions. Although the 1982 paper only covers equality conditions, many RETE implementations support other operations. For example, `Account.name == "Mr Trout"` is a literal condition. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode. In Dr. Forgy's paper, he refers to these as IntraElement conditions. The following shows the AlphaNode combinations for `Cheese(name == "cheddar", strength == "strong")`:

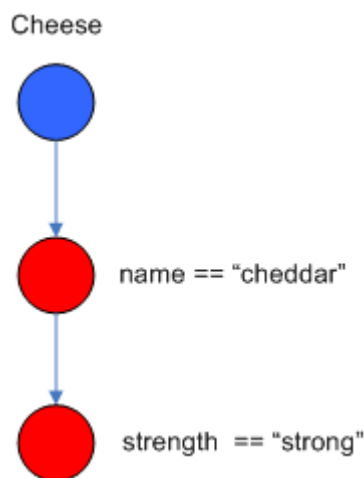


Figure 1.7. AlphaNodes

Drools extends Rete by optimising the propagation from ObjectTypdeNode to AlphaNode using hashing. Each time an AlphaNode is added to an ObjectTypdeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectTypde node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap - avoiding unnecessary literal checks.

There are two two-input nodes; JoinNode and NotNode - both are types of BetaNodes. BetaNodes are used to compare 2 objects, and their fields, to each other. The objects may be the same or different types. By convention we refer to the two inputs as left and right. The left input for a BetaNode is generally a list of objects; in Drools this is a Tuple. The right input is a single object. Two Nots can be used to implement 'exists' checks. BetaNodes also have memory. The left input is called the Beta Memory and remembers all incoming tuples. The right input is called the Alpha Memory and remembers all incoming objects. Drools

extends Rete by performing indexing on the BetaNodes. For instance, if we know that a BetaNode is performing a check on a String field, as each object enters we can do a hash lookup on that String value. This means when facts enter from the opposite side, instead of iterating over all the facts to find valid joins, we do a lookup returning potentially valid candidates. At any point a valid join is found the Tuple is joined with the Object; which is referred to as a partial match; and then propagated to the next node.

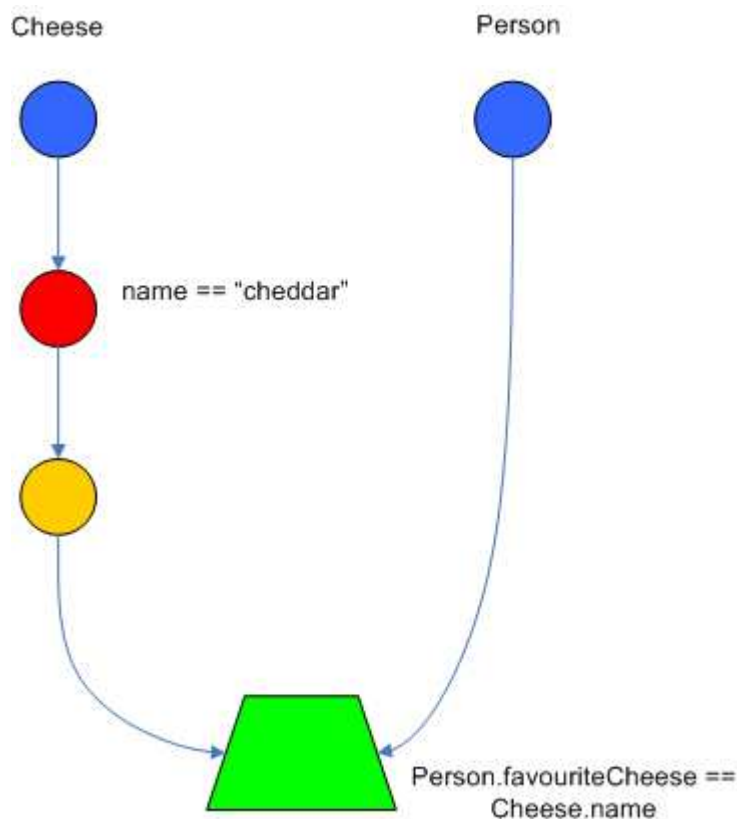


Figure 1.8. JoinNode

To enable the first Object, in the above case Cheese, to enter the network we use a LeftInputNodeAdapter - this takes an Object as an input and propagates a single Object Tuple.

Terminal nodes are used to indicate a single rule has matched all its conditions - at this point we say the rule has a full match. A rule with an 'or' conditional disjunctive connective results in subrule generation for each possible logically branch; thus one rule can have multiple terminal nodes.

Drools also performs node sharing. Many rules repeat the same patterns, node sharing allows us to collapse those patterns so that they don't have to be re-evaluated for every single instance. The following two rules share the first same pattern, but not the last:

```

rule
when
    Cheese( $chedddar : name == "cheddar" )
    $person : Person( favouriteCheese == $chedddar )
then
    System.out.println( $person.getName() + " likes cheddar" );
end
  
```

```

rule
when
    Cheese( $chedddar : name == "cheddar" )
  
```

```
$person : Person( favouriteCheese != $cheddar )  
then  
  System.out.println( $person.getName() + " does not like cheddar" );  
end
```

As you can see below, the compiled Rete network shows the alpha node is shared, but the beta nodes are not. Each beta node has its own TerminalNode. Had the second pattern been the same it would have also been shared.

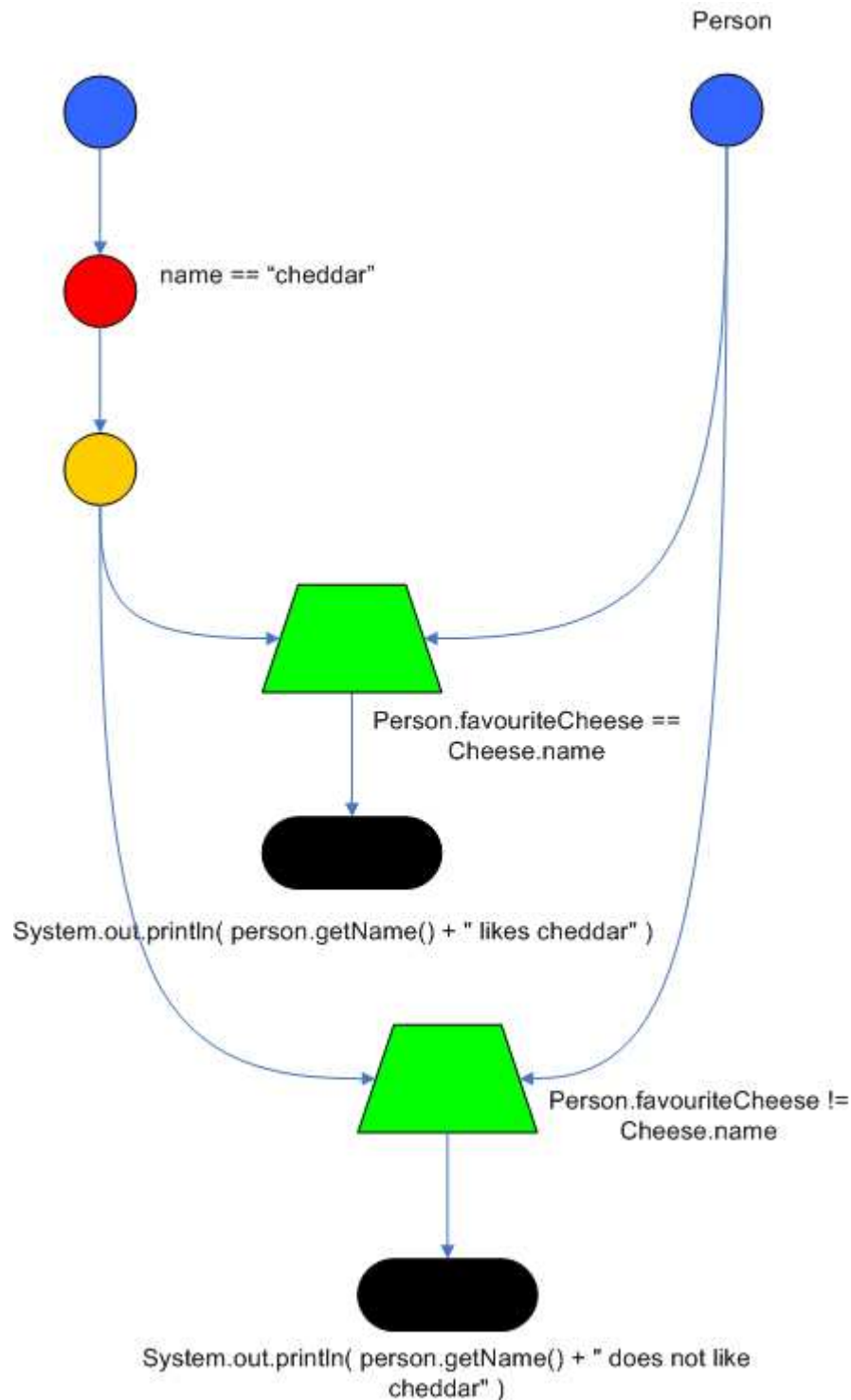


Figure 1.9. Node Sharing

[Prev](#)[Up](#)[Next](#)[1.3. Knowledge Representation](#)[Home](#) | [ToC](#)[1.5. Leaps Algorithm](#)