# An Introduction to the Rete Algorithm

# What is Rete?

- A public domain, efficient pattern matching algorithm
- Initially published by Dr. Charles Forgy in his 1979 Ph.D. thesis
- The basis of most modern inference rule engines (CLIPS, Jess, JBoss Rules/Drools, ILOG JRules, Fair Isaac Blaze Advisor, etc.)
- Pronunciation: 'REET', 'REE-tee', or more commonly in Europe 're-tay', after the Latin rete, meaning network (from Wikipedia)

# Purpose of Rete

- From Dr. Forgy's Thesis: "Production systems have historically operated from one to two orders of magnitude slower than conventional programs, due in large part to the difficulty of performing the match."
- Matching inefficiency in previous algorithms increased as a factor of both the number of rule conditions and the number of objects in WM making large systems impractical
- Shorten execution times of pattern matching when updating the agenda

# Assumptions of Rete

- Desired rule execution behavior is 'inference' or 'rule-chaining' rather than 'sequential'
- Working Memory changes slowly compared to pattern matching cycle times
- Pattern matching involves comparisons that are expensive to repeatedly reproduce. Rules tend to share conditional comparisons

# Assumptions of Rete

- Our rule systems are sufficiently complex enough that network set-up time will be compensated by improved matching performance
- We are willing to trade additional memory consumption for execution speed improvement

# Basics of Rete

- A directed acyclic graph or DAG
- A stateful network of interconnected nodes (stateful of both with regard to WM and to rule conditions)
- Represents the entire, active rule set and 'current state' of objects in WM that may induce a change in the agenda

# Basics of Rete

- Two distinct parts to the network:
  - Alpha network (left side): a discrimination network. Conditions involving only individual attributes of WM elements
  - Beta network (right side): implements join conditions between attributes of different WM elements

# Basics of Rete

- A single entry point where changes to Working Memory are fed.
- Insertion is represented by a positive token; retraction by a negative token. Update is logically a retract (old) and insert (new). Tokens may split at forks.
- Each path terminates at a node representing a single rule in the rule set. A token reaching a exit point induces an agenda change for that rule.

# Rule Set Example

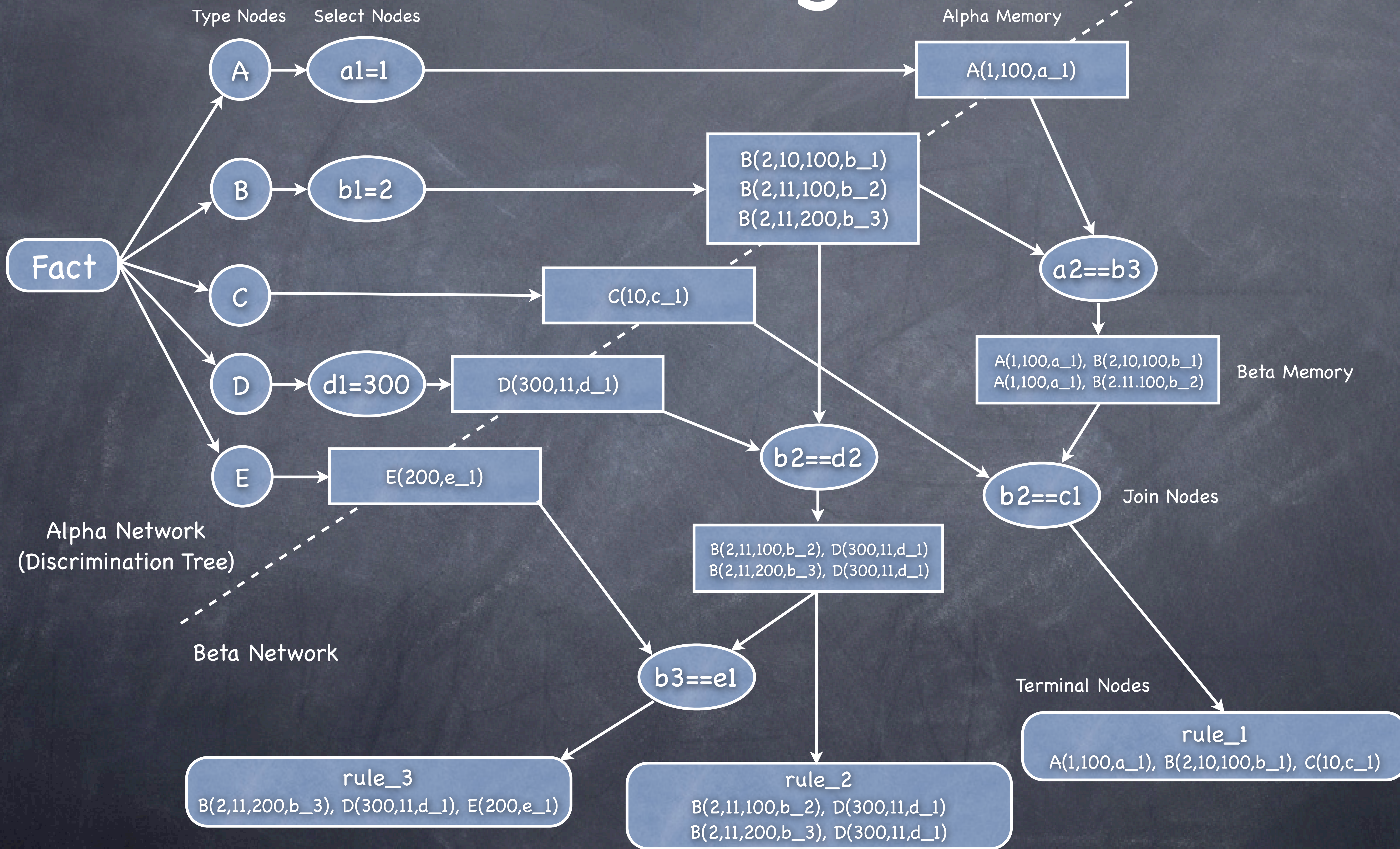
```
rule "rule_1"
    when
        A( a1 == 1, $x: a2 )
        B( b1 == 2, $y: b2, b3 == $x )
        C( c1 == $y )
    then
        System.out.println( "rule_1" );
end
rule "rule_2"
    when
        B( b1 == 2, $y: b2 )
        D( d1 == 300, d2 == $y )
    then
        System.out.println( "rule_2" );
end
```

```
rule "rule_3"
    when
        B( b1 == 2, $y: b2, $z: b3 )
        D( d1 == 300, d2 == $y )
        E( e1 == $z )
    then
        System.out.println( "rule_3" );
end
```

# Example Facts

A (a1 = 1, a2 = 100, "a_1")
A (a1 = 2, a2 = 100, "a_2")
B (b1 = 2, b2 = 10, b3 = 100, "b_1")
B (b1 = 2, b2 = 11, b3 = 100, "b_2")
B (b1 = 2, b2 = 11, b3 = 200, "b_3")
C (c1 = 10, "c_1")
D (d1 = 300, d2 = 11, "d_1")
E (e1 = 200, "e_1")

# Rete Diagram



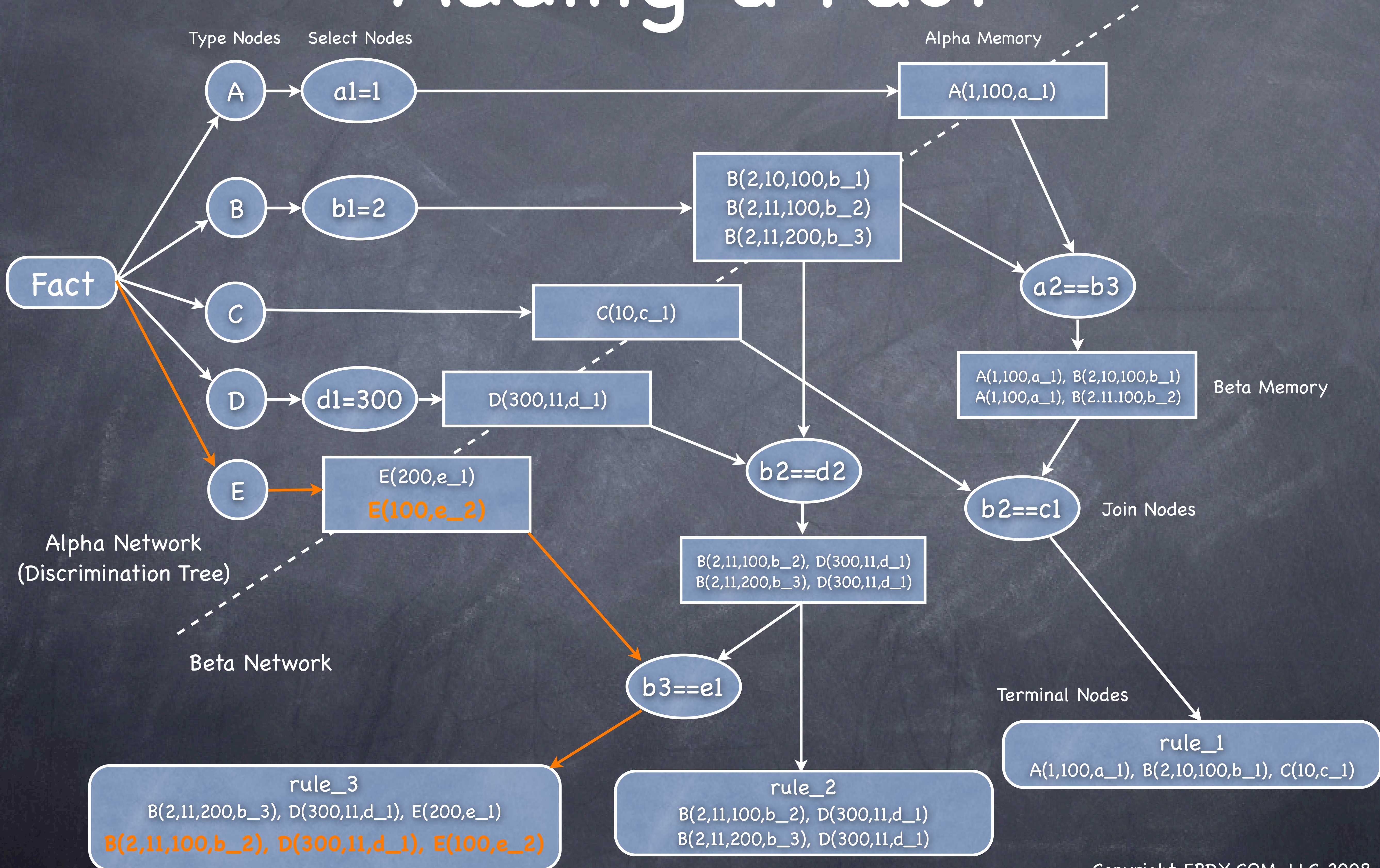
# Agenda

rule_1: A(1, 100, a_1), B(2, 10, 100, b_1), C(10, c_1)
rule_2: B(2, 11, 100, b_2), D(300, 11, d_1)
rule_2: B(2, 11, 200, b_3), D(300, 11, d_1)
rule_3: B(2, 11, 200, b_3), D(300, 11, d_1), E(200, e_1)

# Insert Fact

E (e1 = 100, "e_2")

# Adding a Fact



# Updated Agenda

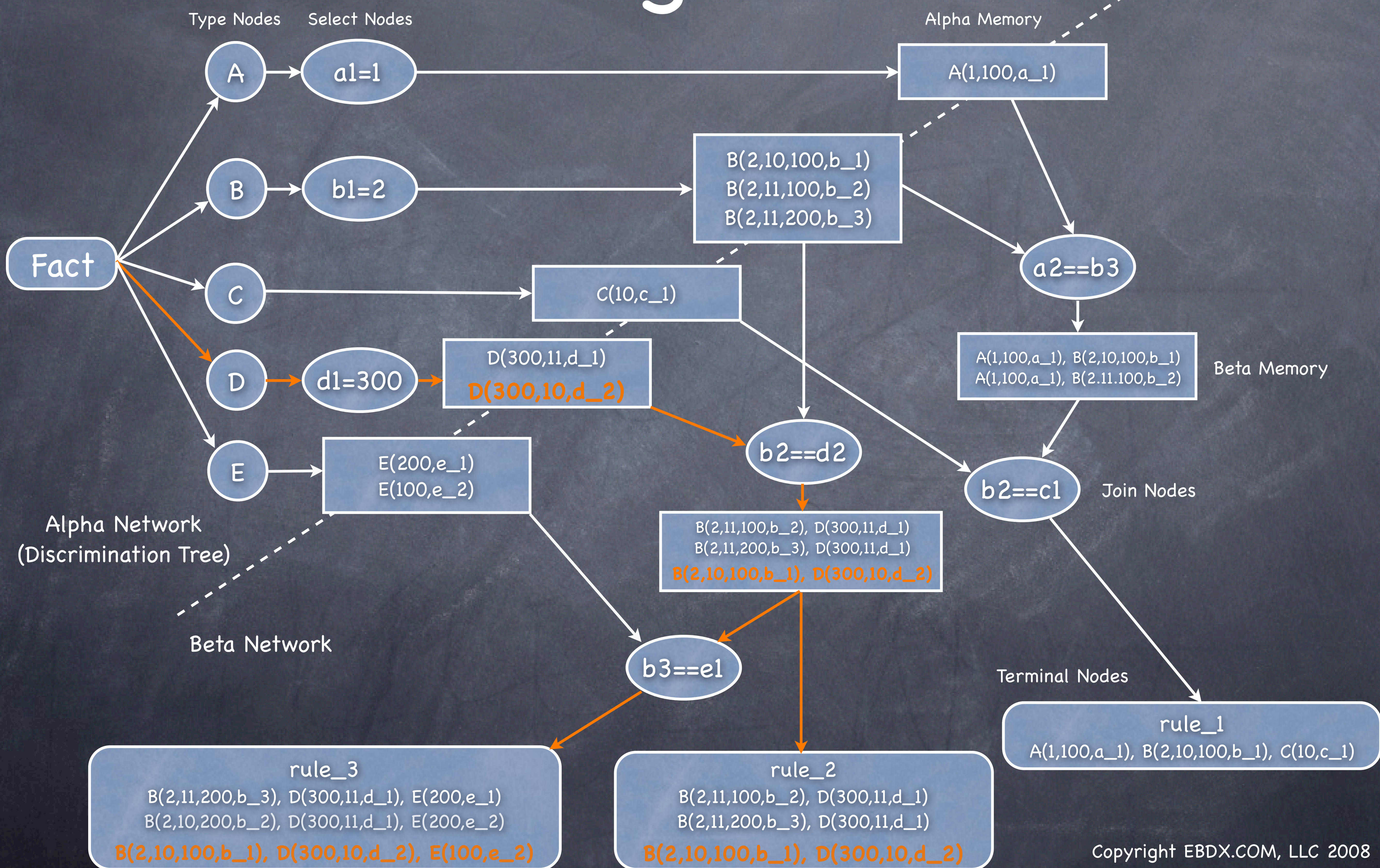
- **rule_3: B(2,11,100,b_2), D(300,11,d_1), E(100,e_2)**

rule_1: A(1,100,a_1), B(2,10,100,b_1), C(10,c_1)
rule_2: B(2,11,100,b_2), D(300,11,d_1)
rule_2: B(2,11,200,b_3), D(300,11,d_1)
rule_3: B(2,11,200,b_3), D(300,11,d_1), E(200,e_1)

# Insert Fact

D (d1 = 300, d2 = 10, "d_2")

# Adding a Fact



# Updated Agenda

- **rule_2: B(2,10,100,b_1), D(300,10,d_2)**
- **rule_3: B(2,10,100,b_1), D(300,10,d_2), E(100,e_2)**

rule_3: B(2,11,100,b_2), D(300,11,d_1), E(100,e_2)
rule_1: A(1,100,a_1), B(2,10,100,b_1), C(10,c_1)
rule_2: B(2,11,100,b_2), D(300,11,d_1)
rule_2: B(2,11,200,b_3), D(300,11,d_1)
rule_3: B(2,11,200,b_3), D(300,11,d_1), E(200,e_1)

# Practical Implications

- Multiple rules sharing the same condition only require the condition to be re-evaluated once, each time the observed attribute changes
- Traversal depth is a factor of the number of conditions on a given rule, not the number of rules within the rule set
- WM change notification is critical. Unnecessary WM change notification is not expensive, but should be avoided when possible

# Questions?

Larry Terrill
EBDX.COM
rules@ebdx.com