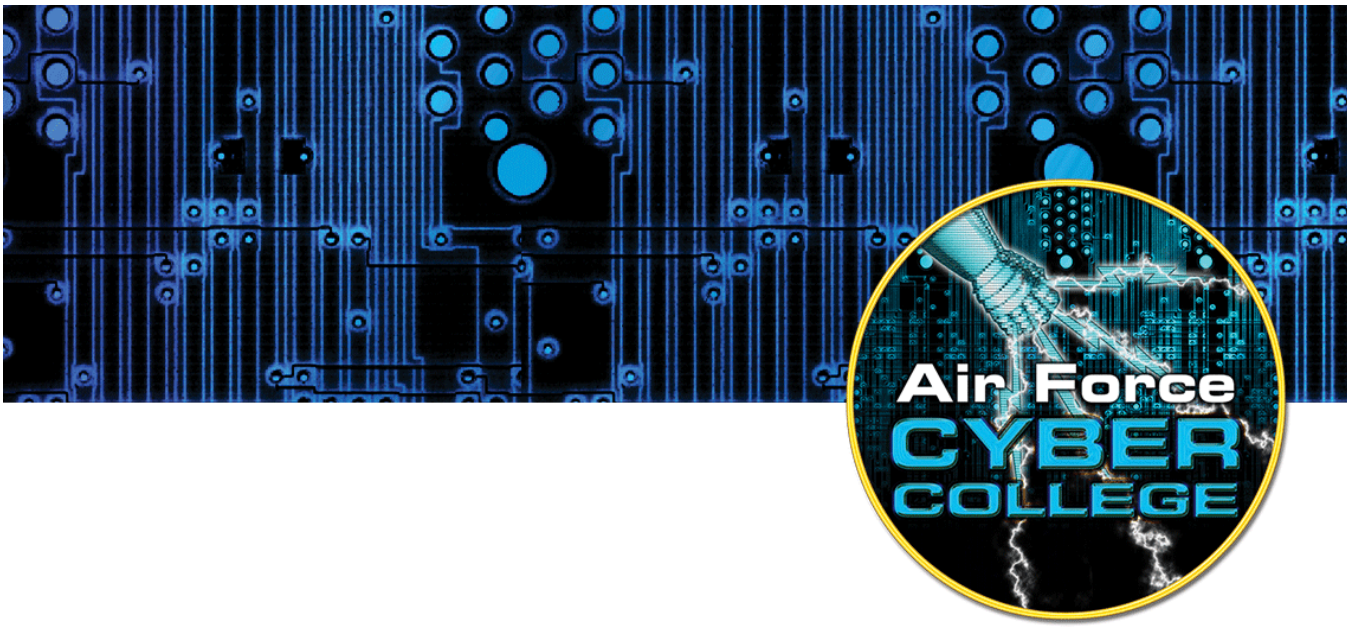


Hands-on Keyboard: Cyber Experiments for Strategists and Policy Makers

Boolean Algebra and Access Control Logic

1. Introduction and Objectives

This exercise covers the basics of the binary and hexadecimal number systems, as well as different methods for binary and hexadecimal conversion. In order to understand Access Control Logic, we introduce Boolean Logic, Set Theory, and Kripke Structures.

The objectives of this exercise are to

- Understand the binary and hexadecimal number systems and be able to convert between the two.
- Understand Boolean Logic.
- Be able to complete Set Theory proofs and relations.
- Understand the basics of Access Control Logic.

2. Binary and Hexadecimal Number Systems

We are all familiar with counting in the decimal system. Although we may not realize it, counting in binary and hexadecimal works in the same manner, the difference being binary has 2 unique symbols while hexadecimal has 16. The quantity of symbols in a numbering system corresponds to the base of the system. Binary (base two) counts in powers of 2, whereas

The Air Force Cyber College thanks the Advanced Cyber Engineering program at the Air Force Research Laboratory in Rome, NY, for providing the information to assist in educating the general Air Force on the technical aspects of cyberspace.

hexadecimal (base sixteen) counts in powers of 16. This makes sense when we take into account that decimal (base 10) counts in powers of 10.

2.1. Binary

The binary number system consists solely of 1s and 0s. Each binary digit is a bit. Each bit is a power of 2. The table below shows the values of the powers of 2 starting with the 0^{th} bit and ending with the 7^{th} bit.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

We can represent the positive numbers 0–127 with this 8-bit table. Note: 128 cannot be represented because we begin counting at 0, not 1. Hence, the largest positive number we can represent given n bits is $2^n - 1$.

2.2. Binary to Decimal Conversion

In order to convert a number from binary to decimal, we add the power of two that corresponds to the location of a 1 in the binary representation.

For example, to convert the binary representation 1010 to decimal, for each 1, calculate the corresponding power of 2 and sum them. Ignore all 0's. The chart below breaks down each step of the conversion process.

Binary	1	0	1	0
Exponent	2^3	2^2	2^1	2^0
Calculated	8	4	2	1
Action	Add 8	Ignore	Add 2	Ignore
Decimal	10			

In the above table, we added 8 and 2 together because their bit location contain a 1. This results in the decimal number 10, which is indeed the number represented. The following table shows the binary values for the decimal numbers 0-10.

Binary	Exponent	Decimal
0000	$0 + 0 + 0 + 0$	0
0001	$0 + 0 + 0 + 2^0$	1

010	$0 + 0 + 2^1 + 0$	2
011	$0 + 0 + 2^1 + 2^0$	3
100	$0 + 2^2 + 0 + 0$	4
101	$0 + 2^2 + 0 + 2^0$	5
110	$0 + 2^2 + 2^1 + 0$	6
111	$0 + 2^2 + 2^1 + 2^0$	7
000	$2^3 + 0 + 0 + 0$	8
001	$2^3 + 0 + 0 + 2^0$	9
010	$2^3 + 0 + 2^1 + 0$	10

As a side note, a short cut to read shorter binary numbers is to read the sequence from left to right. The numbers are summed starting from 0. The first 1 is kept as the current sum. With each subsequent 0, double the previous sum. For each 1 double the previous sum and add 1. For example, given the binary string 01001, reading from left to right, the table below demonstrates how to calculate the correct decimal number of 9. The current digit at each step is bold and underlined in the following table:

inary	Notes/Instruction	Previous Sum	Current Sum
1001	Have not hit a 1 yet, so ignore	0	0
1001	Hit first 1 Start adding from 1	0	1
1**0**01	Double previous sum	1	2
10**0**1	Double previous sum	2	4
100**1**	Double previous sum and add 1	4	9

2.3. Hexadecimal

The numbering system we grow up with only has ten unique symbols (0-9). Therefore, to represent hexadecimal numbers, we need additional symbols. We utilize the letters A through F to supply the additional six symbols. Therefore, hexadecimal numbers are represented with the symbols 0 through 9 and A through F where A through F represent the numbers 10 through 15 respectively.

2.4. Hexadecimal to Decimal

In order to convert hexadecimal to decimal, we add the quantity of the numeric value of the hexadecimal value times sixteen to the power of the hexadecimal location for hexadecimal numbers. For example, the following converts the hexadecimal value 2AF3 to decimal:

Hexadecimal	Equivalent Character	Location	Conversion
2	2	3	(2×16^3)
A	10	2	(10×16^2)
F	15	1	(15×16^1)
3	3	0	(3×16^0)

$$2AF3 = (2 \times 16^3) + (10 \times 16^2) + (15 \times 16^1) + (3 \times 16^0) = 10995$$

2.5. Hexadecimal to Binary

Each hexadecimal value represents four binary values. Hexadecimal shortens binary notation as it is a more compact representation. The following chart shows the binary and hexadecimal equivalencies.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6

7	111	7
8	000	8
9	001	9
10	010	A
11	011	B
12	100	C
13	101	D
14	110	E
15	111	F

For example, to convert 1010010101110010 to hexadecimal, we break the binary representation down into groups of four:

1010010101110010 = 1010 0101 0111 0010

Using the chart above or by converting from binary to decimal as we just learned, we can replace each four bits with their hexadecimal representation:

1010 0101 0111 0010 = A 5 7 2 = A572

Using the same method, we can represent the hexadecimal number A4B7F in binary.

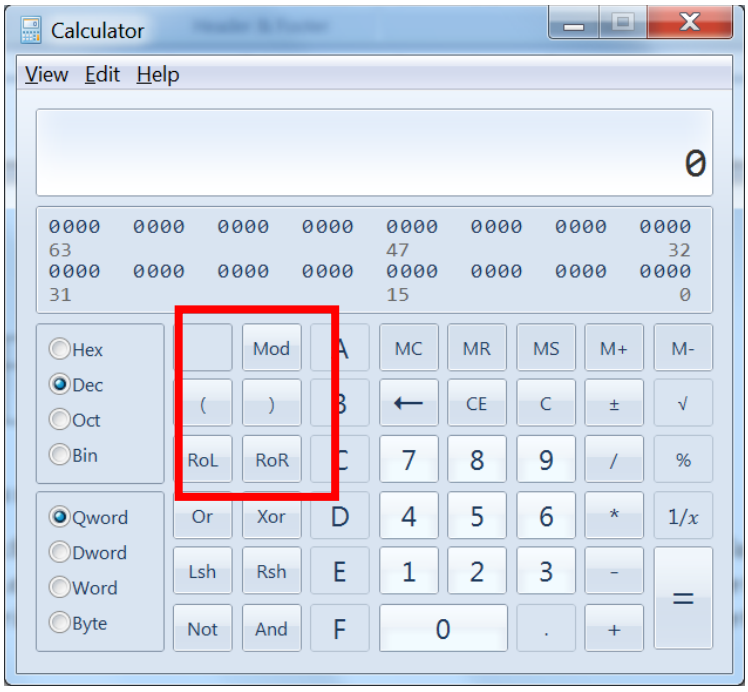
A 4 B 7 F

= 1010 0100 1011 0111 1111

= 10100100101101111111

2.6. Windows Calculator

A useful trick for converting between binary, decimal, and hexadecimal is present in the Windows Calculator. To open the calculator, click ***Start***, type calculator in the search bar, and hit enter. Once the Windows Calculator is open, click ***View*** → ***Programmer***.



To convert a number from one system to another, first select your starting system (Hex, Dec, Oct, or Bin) by clicking on its radial button. Then, type the number you wish to convert from that system. Now, click the radial button for the system to which you want to convert the number.

2.7. Exercises

Convert the following binary numbers into decimal:

1. 10010000
2. 00000001
3. 01110010
4. 00000100
5. 00100100

Convert the following decimal numbers into binary:

1. 63
2. 95
3. 112
4. 46
5. 50

Convert the following hexadecimal numbers into binary and decimal:

1. F4B5C
2. 2A
3. 13C
4. DAD
5. 2C4E

3. Boolean Algebra: Truth Tables and Logic Operators

In Boolean Algebra, a 1 is true and a 0 is false.

A truth table is used to compute the functional values of logical expressions. Negation (¬) produces a value of true if its operand is false and a value of false if the operand is true (NOT p).

Negation	
	p

Logical conjunction (∧) returns true if both values are true (p AND q are true).

Conjunction		
		∧q

Logical disjunction (∨) returns true if at least one operand is true (p OR q is true).

Disjunction		
		∨q

Exclusive disjunction (⊕) produces a value of true if ONLY one operand is true, not both (p XOR q).

Exclusive Disjunction		
		p ⊕q
		0
		1

		1
		0

Negation, conjunction, disjunction, and exclusive disjunction can be combined. Truth tables can be used to determine the value. The truth table below demonstrates the step by step method for calculating $(p \vee \neg q) \oplus (p \wedge q)$.

$(p \vee \neg q) \oplus (p \wedge q)$					
p	q	$\neg q$	$(p \vee \neg q)$	$(p \wedge q)$	$(p \vee \neg q) \oplus (p \wedge q)$
0	0	1	1	0	1
0	1	0	0	0	0
1	0	1	1	0	1
1	1	0	1	1	0

4. Sets and Relations

4.1. Definitions and Notation

- A *set* is a collection of well-defined and distinct objects. Set elements are contained in brackets, as in $\{x_1, x_2, \ldots\ x_n\}$. Note: the order of elements in a set does not matter; therefore, $\{x_1, x_2\}$ is the same as $\{x_2, x_1\}$.
- The *empty set* is the set that has no elements. The empty set is denoted as { } or $\emptyset$.
- The set *union* of two sets A and B is the collection of all elements in either A or B. The union of A and B is denoted A ∪ B.
    1. A = {blue, red}
    2. B = {blue, green}
    3. A ∪ B = {blue, red, green}
- The set *intersection* of two sets A and B is the collection of all elements in A and B. The intersection of A and B is denoted A ∩ B.
    4. A = {blue, red}
    5. B = {blue, green}
    6. A ∩ B = {blue}
- The set *difference* of two sets A and B is the collection of all elements in one that are not in the other. The set difference of A and B is denoted A - B. The collection of all elements

in B that are not in A is denoted B - A. Set difference can also be denoted with \ instead of -.

7. A = {blue, red}
8. B = {blue, green}
9. A - B = {red}
10. B - A = {green}

- To indicate that x is or is not an element of a set, S, we write $x \in S$ to say that x is an element of S and we write $x \notin S$ to say that x is not an element of S.
    11. A = {blue, red}
    12. blue ∈ A
    13. green ∉ A
- A set, S, is a *subset* of another set, T, if every element of S is also an element of T. It may be helpful to think of the subset symbol, ⊆, as a "c" for "is contained within." Note: the empty set is a subset of every set and every set is a subset of itself.
    14. S = {1, 2, 4}
    15. T = {1, 2, 4, 7, 10}
    16. S ⊆ T
    17. T ⊆ T
    18. ∅ ⊆ T
- The *power set* of a set, S, is the set of all subsets of S. The power set is denoted Ƥ(S).
    19. S = {1, 2, 4}
    20. Ƥ(S) = { { }, {1}, {2}, {4}, {1, 2}, {1,4}, {2, 4}, {1, 2, 4} }
- The *Cartesian product* of sets A and B is the set of ordered pairs whose first component is drawn from A and whose second component is drawn from B. Cartesian products are denoted A × B. Note: in set notation, the pipe, "|," reads "such that" or "given that."
    21. $A \times B = \{(a, b) \mid a \in A, b \in B\}$
    22. A = {1, 3, 5}
    23. B = {2, 4, 6}
    24. $A \times B = \{(1,2), (1,4), (1,6), (3,2), (3,4), (3,6), (5,2), (5,4), (5,6)\}$
- A *binary relation* is a set $R \subseteq A \times B$ of pairs whose first components are drawn from *A* and whose second components are drawn from *B*. *R* is a *binary relation over (or on) A* when $R \subseteq A \times A$.
- The *composition* of relations $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times C$ is the relation $R_1 \circ R_2 \subseteq A \times C$ as defined as follows:
    25. $R_1 \circ R_2 = \{(x, z) | there\ exists\ y\ such\ that\ ((x, y) \in R_1 and\ (y, z) \in R_2)\}$
- The *image of R under a* where R is the relation $R \subseteq A \times B$ and $a \in A$ is the set of elements in *B* related to a by the relation *R*.
    26. $R(a) = \{b \in B \mid (a, b) \in R\}$
    27. $R = \{(1,2), (2,3), (2,4), (3,5), (3,1), (4,1), (5,2)\}$
    28. $R(2) = \{3, 4\}$

29. $R(3) = \{5, 1\}$

4.2. Examples and Exercises

4.2.1. Example

1. Let A = {1, 2, 3, 4, 5} and B = {0, 3, 6}.
    a. A ∪ B = {0, 1, 2, 3, 4, 5, 6}
    b. A ∩ B = {3}
    c. A - B = {1, 2, 4, 5}
    d. B - A = {0, 6}
2. Let A = {1, 2} and B = {a, b, c}.
    a. A × B = {(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)}
    b. B × A = {(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)}

4.2.2. Exercises

1. Let A = {a, b, c, d, e} and B = {a, b, c, d, e, f, g, h}. Find
    a. A ∪ B
    b. A ∩ B
    c. A - B
    d. B - A
2. Let T and U be relations over the set A = {1, 2, 3, 4}, as follows: $T = \{(1, 1), (2, 1), (3, 3), (4, 4), (3, 4)\}$ and $U = \{(2, 4), (1, 3), (3, 3), (3, 2)\}$.
    a. Ƥ({x, y, z})
    b. U(3)
    c. U(4)
    d. T ∪ U
    e. T ∩ U
    f. T - U
    g. U ∘ T
    h. T ∘ U
3. Let A = {a, b, c}, B = {x ,y}, and C = {0 ,1}. Find
    a. A × B × C
    b. C × B × A
    c. C × A × B
    d. B × B × B

4.3. Approaches for Mathematical Proofs

4.3.1. Properties

- *Transitivity:* If A is a set such that x ∈ A and y ∈ x then y ∈ A. For example, an apple is a type of edible fruit. Edible fruits are food. Therefore, an apple is food.
- *Equivalence:* Two sets are equal provided that each is a subset of the other. That is, if A and B are sets and A ⊆ B and B ⊆ A then A = B. Colloquially, A has the exact same elements as B.

4.3.2. Example Proof

Prove the following:

If A, B, and C are sets then A ∩ (B ∩ C) = (A ∩ B) ∩ C.

Since we are claiming the equivalence of two sets, we must show that each set is a subset of the other. This gives us two steps: ① shows A ∩ (B ∩ C) ⊆ (A ∩ B) ∩ C, and ② shows (A ∩ B) ∩ C ⊆ A ∩ (B ∩ C). As stated in Section 4.1, a set is a subset of another if every element in the smaller set is in the larger set. To show that A ∩ (B ∩ C) ⊆ (A ∩ B) ∩ C, choose an arbitrary element x ∈ A ∩ (B ∩ C) and show that x ∈ (A ∩ B) ∩ C.

① Show $A \cap (B \cap C) \subseteq (A \cap B) \cap C$.

Let $x \in A \cap (B \cap C)$. Show $x \in (A \cap B) \cap C$.

$\Rightarrow x \in A \cap (B \cap C)$

$\Rightarrow x \in A$ and $x \in (B \cap C)$ by the definition of set intersection

$\Rightarrow x \in A$ and $x \in B$ and $x \in C$ by the definition of set intersection

$\Rightarrow x \in A \cap B$ and $x \in C$

$\Rightarrow x \in (A \cap B)$ and $x \in C$

$\Rightarrow x \in (A \cap B) \cap C$

By showing that any arbitrary x in $A \cap (B \cap C)$ is also in $(A \cap B) \cap C$, we have shown that $A \cap (B \cap C) \subseteq (A \cap B) \cap C$.

② Show $(A \cap B) \cap C \subseteq A \cap (B \cap C)$.

Let $x \in (A \cap B) \cap C$. Show $x \in A \cap (B \cap C)$.

$\Rightarrow x \in (A \cap B) \cap C$

$\Rightarrow x \in (A \cap B)$ and $x \in C$ by the definition of set intersection

$\Rightarrow x \in A$ and $x \in B$ and $x \in C$ by the definition of set intersection

$\Rightarrow x \in A$ and $x \in B \cap C$

$\Rightarrow x \in A$ and $x \in (B \cap C)$

$\Rightarrow x \in A \cap (B \cap C)$

By showing that any arbitrary x in $(A \cap B) \cap C$ is also in $A \cap (B \cap C)$, we have shown that $(A \cap B) \cap C \subseteq A \cap (B \cap C)$.

Since each set is a subset of the other, we can conclude that $A \cap (B \cap C) = (A \cap B) \cap C$.

5. Syntax

5.1. Principal Expressions

Principals are the major actors in a system. The class of principals includes (but is not limited to) people, processes, cryptographic keys, personal identification numbers (PINs), userID-password pairs, and so on. The following are all allowable principal names: *Alice, Bob,* the key K_{Alice}, the PIN 1234, and the userID-password pair ⟨*Alice, bAdPsWd!*⟩.

Compound principals connote a combination of principals. For example, "the President in conjunction with Congress" connotes a principal comprising of both the president and Congress. Also, "the reporter quoting her source" connotes a principal that comprises both the reporter and her source.

A principal expression is a name, an expression of the form *P & Q* or an expression of the form *P* | *Q*. The principal expression *P & Q* denotes the principal "*P* in conjunction with *Q.*" The principal expression *P* | *Q* denotes the principal "*P* quoting *Q.*" Parentheses can be added to disambiguate compound principal expressions. For example, *(Sal & Ted)* | *Uly* denotes the conjunctive principal *Sal & Ted* quoting the principal *Uly*. In contrast, *Sal & (Ted* | *Uly)* denotes the principal *Sal* in conjunction with the principal *Ted* quoting the principal *Uly*. The standard convention in such expressions is that & binds more tightly than |, so *Sal & Ted* | *Uly* is equivalent to (*Sal & Ted)* |*Uly*.

5.2. Access Control Statements

We must determine with precision and accuracy which access requests from which principals should be granted and which should be denied. We need to be able to express our assumptions and our expectations as to which authorities we trust, which principals should be granted access to which objects, and so on. We represent basic requests such as "read file foo" or "modify file bar" by propositional variables. We need some way of accounting for the source of the request. In our logic, we can associate requests with their source by statements of the form

P says φ

where P is the principal and φ is a specific statement. For example, if *rff* is a propositional variable representing the request "read file *foo*," then we can represent Deena's request to read file *foo* by the statement

Deena says *rff*.

The *says* operator can also ascribe non-request statements to particular principals. For example, we may wish to express that Rob believes that Deena is making a request to read file *foo*. We can express this by the statement

Rob says (*Deena* says *rff*).

Access policies specify which principals are authorized to access particular objects. Such authorizations can be expressed in our logic by statements of the form

$$P \text{ controls } \varphi.$$

where P is the principal and φ is a specific statement. For example, we can express Deena's entitlement to read the file *foo* as the statement

$$\textit{Deena} \text{ controls } \textit{rff}.$$

Like authorizations, jurisdiction is represented by statements of the form

$$P \text{ controls } \varphi$$

where P is a principal with jurisdiction over the statement φ.

In order to make statements about the relative trust level of different principals, we use the operator $\Rightarrow$ (pronounced "speaks for"). The statement

$$P \Rightarrow Q$$

describes a proxy relationship between the two principals P and Q such that any statement made by P can also be safely attributed to Q.

We also make use of the standard logical operators: negation ($\neg\varphi$), conjunction ($\varphi_1 \wedge \varphi_2$), disjunction ($\varphi_1 \vee \varphi_2$), implication ($\varphi_1 \supset \varphi_2$), and equivalence ($\varphi_1 \equiv \varphi_2$). According to standard conventions, $\neg$ binds the most tightly, followed in order by $\wedge$, $\vee$, $\supset$ and $\equiv$.

5.3. Well-Formed Formulas

Principal names and propositional variables will be distinguished by capitalization. Specifically, we use capitalized identifiers such as *Josh* and *Reader* for simple principal names. We use lowercase identifiers such as *r, write*, and *rff* for propositional variables.

5.3.1. Backus-Naur Form (BNF)

Backus-Naur Form is a notation used to describe context-free grammars. Those of you familiar with compilation, programming, or protocols are already familiar with context-free grammars, even if you do not realize it.

The set **Form** of all well-formed expressions in our language is given by the following BNF specification:

Form ::= **PropVar** / ¬**Form** / (**Form** ∨ **Form**) / (**Form** ∧ **Form**) / (**Form** ⊃**Form**) / (**Form** ≡ **Form**) / (**Princ** ⇒**Princ**) / (**Princ** says **Form**) / (**Princ** controls **Form**)

We define a nonterminal statement as a sequence of terminal or nonterminal statements. The above BNF specification provides all the information necessary to determine the structure of well-formed formulas in access control logic.

5.3.2. Examples

The following are well-formed formulas:

- r
- $((\neg q \wedge r) \supset s)$
- (*Jill* says $(r \supset (p \vee q))$)

The following syntactic derivation demonstrates that $(Jill\ says\ (r\ \supset (p \vee q)))$ is a well-formed formula:

Form ⇝ (**Princ** says **Form**)
⇝ (**PName** says **Form**)
⇝ (*Jill* says **Form**)
⇝ (*Jill* says (**Form** ⊃ **Form**)
⇝ (*Jill* says (**PropVar** ⊃ **Form**)
⇝ (*Jill* says (r ⊃ **Form**)
⇝ (*Jill* says (r ⊃ (**Form** ∨ **Form**)))
⇝ (*Jill* says (r ⊃ (**PropVar** ∨ **Form**)))
⇝ (*Jill* says (r ⊃ (p ∨ **Form**)))
⇝ (*Jill* says (r ⊃ (p ∨ **PropVar**)))
⇝ (*Jill* says (r ⊃ (p ∨ q)))

The following examples are *not* well-formed formulas, for the reasons stated:

- *Orly & Mitch* is a principal expression but not an access-control formula.
- ¬*Orly*, because *Orly* is a principal expression but not an access-control formula. The negation operator ¬ must precede an access-control formula.
- (*Orly* ⇒ ($p \wedge q$)) because ($p \wedge q$) is not a principal expression. The "speaks for" operator ⇒ must appear between two principal expressions.
- (*Orly* controls *Mitch*) because *Mitch* is a principal expression, not an access-control formula. The controls operator requires its second argument to be an access-control formula.

5.4. Exercises

1. Which of the following are well-formed formulas in the access-control logic? Support your answers by appealing to the BNF specification.
   a. $((p \wedge \neg q) \supset$ (*Cal* controls r))
   b. ((*Gin* ⇒ r) ∧ q)
   c. (*Mel* | *Ned* says ($r \supset t$))
   d. ($\neg t$ ⇒ *Sal*)
   e. (*Ulf* controls (*Vic* | *Wes* ⇒ *Tori*))
   f. (*Pat* controls (*Quint* controls (*Ryne* says s)))
2. Fully parenthesize each of the following formulas:
   a. $p \supset \neg q \vee r \supset s$

b. ¬p ⊃ r ≡ q ∨ r ⊃ t
c. *X* controls *t* ∨ *s* ⊃ *Y* says *q* ⊃ *r*
d. *Cy* controls *q* ∧ *Di* controls *p* ⊃ *r*
e. Ike ⇒ Jan ∧ Kai & Lee controls q ∧ r

6. Kripke Structures

6.1. Definition

A *Kripke structure*, M is a three-tuple ⟨W, I, J⟩, where:

- W is a nonempty set, whose elements are called worlds
- I: **PropVar** →℘(W) is an interpretation function that maps each propositional variable to a set of worlds
- J: **PName**→℘(W ×W) is a function that maps each principal name to a relation on worlds (i.e. a subset of W × W).

The concept of worlds is abstract. In reality, W is simply a set. The functions I and J provide meanings for our propositional variables and simple principals. *I(p)* is the set of worlds in which we consider *p* to be true. *J(A)* is a relation that describes how the simple principal *A* views the relationships between worlds. Each pair *(w, w′)* ∈ *J(A)* indicates that when the current world is *w*, principal *A* believes it possible that the current worlds is *w′*.

6.2. Example

Consider three young children (*Flo*, *Gil,* and *Hal*) who are being watched by an overprotective babysitter. The babysitter will let them go outside to play only if the weather is both sunny and warm. Imagine there are only three possible situations: it is sunny and warm, it is sunny but cool, or it is not sunny. We represent these as a set of three worlds: $W_0 = \{sw, sc, ns\}$. We use the propositional variable *g* t o represent the proposition "The children can go outside." The babysitter's overprotectiveness can be represented by any interpretation function: I_0: **PropVar** → ℘({sw, sc, ns}) for which $I_0(g) = \{sw\}$. That is, the proposition *g* is true only in the world *sw*.

The children are standing by the window trying to determine whether or not they will be allowed to go outside. *Gil* is tall enough to see the outdoor thermometer and possesses perfect knowledge of the situation. This corresponds to a possible-worlds relation $J_0(Gil) = \{(sw, sw), (sc, sc), (ns, ns)\}$. *Flo* is too short to see the outdoor thermometer and cannot determine between "sunny and warm" and "sunny and cool." This corresponds to a possible-worlds relation $J_0(Flo) = \{(sw, sw), (sw, sc), (sc, sw), (sc, sc), (ns, ns)\}$. That is, $J_0(Flo)(sw) = \{sw, sc\}$. *Hal* is too young to understand that it can be simultaneously sunny and cool. He believes that the presence of the sun automatically makes it warm outside. His confusion corresponds to a possible-worlds relation $J_0(Hal) = \{(sw, sw), (sc, sw), (ns, ns)\}$. That is, $J_0(Hal)(sc) = \{sw\}$. The tuple $\langle W_0, I_0, J_0 \rangle$ forms a Kripke structure.

6.3. Exercise

In addition to the Kripke structure above, suppose that

$J_0(Ida)$ = {(sw, sc), (sc, sw), (ns, sc), (ns, ns)}.

Calculate the following relations:

1. J_0(Hal & Gil)
2. J_0(Gil | Hal)
3. J_0(Flo & Ida)
4. J_0(Hal | Ida)
5. J_0(Ida | Hal)
6. J_0(Hal & (Ida | Hal))
7. J_0(Hal | (Ida & Hal))

7. Access Control Logic

7.1. Logical Rules

Each rule of logic has the following form:

$$\frac{H_1 \ \ldots \ H_k}{C}$$

where H_i is a hypothesis and C is the conclusion or consequence. Informally, this is read if each H above the line is true then we can conclude C below the line. It is possible to have no hypothesis (i.e. k = 0). These cases are called axioms. Each rule states that, if all the premises of an inference rule have already been written derived then the conclusion can also be derived. Axioms can always be derived.

7.1.1. The *Taut* Rule

A propositional-logic tautology is a formula that evaluates to *true* under *all* possible interpretations of its propositional variables.

$$Taut\ \frac{}{\varphi}\ \text{if } \varphi \text{ is an instance of a prop} - \text{logic tautology}$$

This axiom states that any instance of a *tautology* from propositional logic can be introduced at any time as a derivable statement in the access control logic.

For example, the formula

(*Alice* says *go*) ∨ ((*sit* ∧ *read*) ⊃ (*Alice* says *go*))

is an instance of the formula $q \vee (r \supset q)$ since it can be obtained by replacing every q by (*Alice* says *go*) and every r by (*sit* ∧ *read*). In contrast, the formula

(*Alice* says *go*) ∨ ((*sit* ∧ *read*) ⊃ *stay*)

is *not* an instance of the formula $q \vee (r \supset q)$, because the two separate occurrences of q were not replaced by the same formula.

7.1.2. The Modus Ponens Rule

$$Modus\ Ponens\ \frac{\varphi \qquad \varphi \supset \varphi'}{\varphi'}$$

This rule states that if both the implication φ ⊃ φ' and the formula φ have been previously introduced, then we can also introduce the formula φ'. For example, if we have previously derived the two formulas

$(Bill\ \text{says}\ sell) \supset buy$ and $Bill\ \text{says}\ sell$

then we can use the *Modus Ponens* rule to derive *buy*.

7.1.3. The *Says* Rule

$$Says\ \frac{\varphi}{P\ \text{says}\ \varphi}$$

This rule states that any principal can make any statement (or safely be assumed to have made that statement) that has already been derived. For example, if we have previous derived (*read* ∧ *copy*), then we can derive *Cara* says (*read* ∧ *copy*).

7.1.4. The *MP Says* Rule

$$MP\ Says\ \frac{}{\left(P\ \text{says}\ (\varphi\ \supset \varphi')\right) \supset (P\ \text{says}\ \varphi\ \supset P\ \text{says}\ \varphi'}$$

This rule allows us to distribute the says operator over implications. For example, this axiom allows us to derive the following formula:

(*Graham* says (*sit* ⊃ *eat*)) ⊃ ((*Graham* says *sit*) ⊃ (*Graham* says *eat*)).

7.1.5. The Speaks For Rule

$$Speaks\ For\ \frac{}{P\ \Rightarrow Q\ \supset (P\ \text{says}\ \varphi\ \supset Q\ \text{says}\ \varphi)}$$

This rule captures our intuition about the speaks-for relation. It states that if *P* speaks for *Q* then any statements *P* makes should also be attributable to *Q*. For example, this axiom allows us to derive the following statement:

Del ⇒ *Ed* ⊃ ((*Del* says *buy*) ⊃ (*Ed* says *buy*)).

7.1.6. The & Says Rule

$$\&\ Says\ \frac{}{(P\ \&\ Q\ \text{says}\ \varphi) \equiv ((P\ \text{says}\ \varphi) \wedge (Q\ \text{says}\ \varphi))}$$

This rule reflects the conjunctive nature of a principal *P & Q*. The statements made by the compound principal *P & Q* are precisely those statements that both *P* and *Q* are willing to make individually. For example,

(*Faith* & *Gail* says *sing*) ≡ ((*Faith* says *sing*) ∧ (*Gail* says *sing*).

7.1.7. The *Quoting* Rule

$$Quoting\ \frac{}{(P\ |Q\ \text{says}\ \varphi)\ \equiv (P\ \text{says}\ Q\ \text{says}\ \varphi)}$$

This rule captures the underlying intuition behind the compound principal $P \mid Q$. The statements made by $P \mid Q$ are precisely those statements that P claims Q has made. For example,

(Iona | Jill says vote for Kory) ≡ (Iona says Jill says vote for Kory).

7.1.8. Properties of ⇒

The *Idempotency of* ⇒ states that every principal speaks for itself.

$$Idempotency\ of\ \Rightarrow\ \frac{}{P\ \Rightarrow P}$$

For example we can derive the following formula:

Wallace ⇒ Wallace.

The *Transitivity of* ⇒ supports reasoning about chains of principals that represent one another.

$$Transitivity\ of\ \Rightarrow\ \frac{P\ \Rightarrow Q \quad Q\ \Rightarrow R}{P\ \Rightarrow R}$$

For example, if we have previously derived the following two formulas

Kanda ⇒ Theo and Theo ⇒ Vance

then the *Transitivity of* ⇒ rule allows us to derive

Kanda ⇒ Vance.

The *Monotonicity of* ⇒ rule states that quoting principals preserve the speaks-for relationship.

$$Monotonicity\ of\ \Rightarrow\ \frac{P\ \Rightarrow P' \quad Q\ \Rightarrow Q'}{P \mid Q\ \Rightarrow P' \mid Q'}$$

For example, suppose we have already derived the following two formulas:

Lowell ⇒ Minnie Norma ⇒ Orson.

The *Monotonicity of* ⇒ rule allows us to derive the formula

Lowell | Norma ⇒ Minnie | Orson.

7.1.9. The *Controls* Definition

$$P\ \text{controls}\ \varphi\ \stackrel{\text{def}}{=}\ (P\ \text{says}\ \varphi) \supset \varphi$$

Controls does not give our logic any additional expressiveness, but it provides a useful way to make more explicit what will turn out to be a common idiom.

7.2. Example Formal Proofs

An example formal proof:

1. *Al* says $(r \supset s)$	Assumption
2. r	Assumption
3. (*Al* says $(r \supset s)) \supset$ (*Al* says $r \supset$ *Al*	MP Says

says *s*)

4. *Al* says $r \supset$ *Al* says *s*	1,3 Modus Ponens
5. *Al* says *r*	2 Says
6. *Al* says *s*	4,5 Modus Ponens

Formal Proof of the *Controls* rule:

1. *P* controls φ	Assumption
2. *P* says φ	Assumption
3. (*P* says φ) $\supset$ φ	1 Definition controls
4. φ	2,3 Modus Ponens

7.3. Useful Derived Rules

$$Conjunction\ \frac{\varphi_1 \quad \varphi_2}{\varphi_1 \wedge \varphi_2}$$

$$Simplification\ (1)\ \frac{\varphi_1 \wedge \varphi_2}{\varphi_1} \qquad Simplification\ (2)\ \frac{\varphi_1 \wedge \varphi_2}{\varphi_2}$$

$$Disjunction\ (1)\ \frac{\varphi_1}{\varphi_1 \vee \varphi_2} \qquad Disjunction\ (2)\ \frac{\varphi_2}{\varphi_1 \vee \varphi_2}$$

$$Modus\ Tollens\ \frac{\varphi_1 \supset \varphi_2 \quad \neg\varphi_2}{\neg\varphi_1} \qquad Double\ negation\ \frac{\neg\neg\varphi}{\varphi}$$

$$Disjunctive\ Syllogism\ \frac{\varphi_1 \vee \varphi_2 \quad \neg\varphi_1}{\varphi_2} \qquad Hypothetical\ Syllogism\ \frac{\varphi_1 \supset \varphi_2 \quad \varphi_2 \supset \varphi_3}{\varphi_1 \supset \varphi_3}$$

$$Controls\ \frac{P \text{ controls } \varphi \quad P \text{ says } \varphi}{\varphi}$$

$$Derived\ Speaks\ For\ \frac{P \Rightarrow Q \quad P \text{ says } \varphi}{Q \text{ says } \varphi} \qquad Derived\ Controls\ \frac{P \Rightarrow Q \quad Q \text{ controls } \varphi}{P \text{ controls } \varphi}$$

$$Says\ Simplification\ (1)\ \frac{P \text{ says } (\varphi_1 \wedge \varphi_2)}{P \text{ says } \varphi_1} \qquad Says\ Simplification\ (2)\ \frac{P \text{ says } (\varphi_1 \wedge \varphi_2)}{P \text{ says } \varphi_2}$$

7.4. Exercises

7.4.1. Give a formal proof for the *Derived Speaks For* rule.

7.4.2. Give a formal proof for the *Derived Controls* rule.

7.4.3. Give a formal proof for the Says Simplification rule.