2 • Analyzing Syntactic Semantics

Once we have a syntax and a semantics for a programming language, we can ask questions, experiment, and contemplate alternatives. In this part of the book, we look at the most basic questions that programming language theoreticians ask, and we study how to answer them. In the second part of the book, we introduce a tool for experimenting with syntax and semantics, which usually helps formulate conjectures and questions.

Here we use the syntax and semantics introduced in the first chapter to illustrate what kinds of questions to ask and how to answer them rigorously. The first section shows how to formulate questions about a language in mathematical terms. The second section formulates answers as mathematical theorems and proofs, introducing the key proof techniques for the rest of this part of the book.

2·1 From Questions to Mathematical Claims.

The first chapter defines several evaluators, including $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}$. Viewed from the perspective of a language implementor, this function uses something like a machine whose initial, intermediate, and final states are B expressions and whose instructions are the $\rightarrow_{\mathbf{r}}$ relation. It launches the program, waits until the machine reaches a final state (`t` or `f`), and reports this final result.

One obvious question is whether this evaluator always produces exactly one result for some fixed program. In mathematical terms, we are asking whether the evaluator is a function. If so, we know that the implementation of the evaluator is broken if we ever observe two distinct results for one and the same program.

Now recall that, like a relation, a function is a set of pairs; each pair combines an "input" with the "output." The difference between a relation and a function is that the latter contains at most one pair for any input. Thus, our first question means to ask whether the following claim holds:

$$\text{for all } B_0,\ (B_0, R_0) \in eval_{\mathbf{r}}^{=_{\mathbf{r}}} \text{ and } (B_0, R_1) \in eval_{\mathbf{r}}^{=_{\mathbf{r}}} \text{ implies } R_0 = R_1.$$

In functional notation, this becomes

$$\text{for all } B_0,\ eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_0) = R_0 \text{ and } eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_0) = R_1 \text{ implies } R_0 = R_1.$$

Chapter 1 does not just define one evaluator; it actually defines *two*. Ideally, the two definitions should introduce the same function. That is, the

two evaluators should produce identical results when given the same program. Proving such a claim would allow us to use the two definitions interchangeably and according to our needs. For example, while arguing with a mathematics teacher, we could use $eval_{\mathbf{r}}^{=\mathbf{r}}$ to demonstrate that program execution generalizes seventh grade algebra. Then again, we should probably use $eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}}$ when we discuss the semantics of B with a software engineer charged with the implementation of the B programming language.

Thus our second question concerns the relationship between the equational evaluator, $eval_{\mathbf{r}}^{=\mathbf{r}}$, and the directional evaluator, $eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}}$. Specifically, we should be wondering whether they are the same function:

$$eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}} = eval_{\mathbf{r}}^{=\mathbf{r}}$$

Given our understanding of functions as sets of pairs, the question is whether the two sets contain the same elements:

$$\text{for all } B_0 \text{ and } R_0,\ ((B_0, R_0) \in eval_{\mathbf{r}}^{=\mathbf{r}} \text{ if and only if } (B_0, R_0) \in eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}})$$

A rough translation of this statement is

$$\text{for all } B_0 \text{ and } R_0,\ (eval_{\mathbf{r}}^{=\mathbf{r}}(B_0) = R_0 \text{ if and only if } eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}}(B_0) = R_0)$$

meaning if $eval_{\mathbf{r}}^{=\mathbf{r}}$ is defined for B_0 and the output is R_0, then $eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}}$ is defined for B_0 and the output is also R_0, and vice versa.

Naturally, the very phrase is "is defined" suggests yet another question, namely, whether the evaluator produces an output for all possible inputs. This statement has a straightforward mathematical formulation as,

$$\text{for all } B, \text{ there exists an } R \text{ such that } (B, R) \in eval_{\mathbf{r}}$$

Note how we dropped the superscript from $eval_{\mathbf{r}}$ assuming it does not matter which one we use.

2·2 Answers as Theorems.

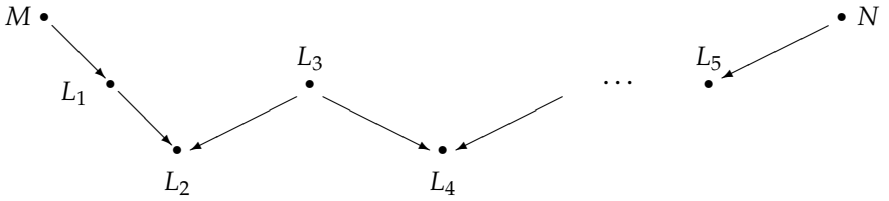
Now that we understand the simplest questions we can ask about the syntax and semantics of a programming language, we state the obvious, namely, that answers are theorems about the mathematical model. Our first theorem states that $eval_{\mathbf{r}}^{=\mathbf{r}}$ is a function.

> **Theorem 2.1**: If $eval_{\mathbf{r}}^{=\mathbf{r}}(B_0) = R_1$ and $eval_{\mathbf{r}}^{=\mathbf{r}}(B_0) = R_2$, then $R_1 = R_2$.

In stating and proving the theorem for $eval_{\mathbf{r}}^{=\mathbf{r}}$, we are following history in first establishing **consistency** for the equational calculus. Once we prove that the two definitions specify the same relation, we also know that $eval_{\mathbf{r}}^{\rightarrow\!\!\!\!\rightarrow\mathbf{r}}$ is a function, too. Also following convention, we drop the quantification prefixes (for all, there exists) from theorems when obvious, and we use functional notation to mean that the result exists and is a specific value.

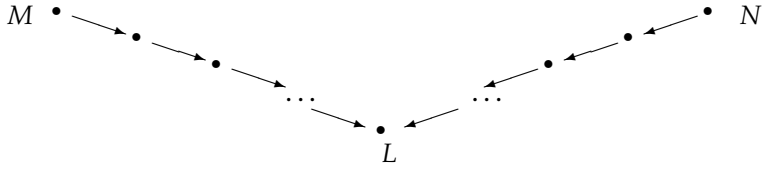
> **Proof for Theorem 2.1**: To prove the theorem, we assume the antecedent and prove that the conclusion follows. That is, we assume that $eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_0) = R_1$ and $eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_0) = R_2$ for some B_0, R_1, and R_2. Based on these assumptions, we now attempt to prove that $R_1 = R_2$. By the definition of $eval_{\mathbf{r}}^{=_{\mathbf{r}}}$, our assumptions imply that $B_0 =_{\mathbf{r}} R_1$ and $B_0 =_{\mathbf{r}} R_2$. Note the use of $=_{\mathbf{r}}$, as opposed to $=$. Because $=_{\mathbf{r}}$ is an equivalence relation (by definition), $R_1 =_{\mathbf{r}} R_2$. To reach the desired conclusion that $R_1 = R_2$, i.e., that R_1 and R_2 are identical, we must study the nature of this equivalence relation and the nature of calculations. (to be continued)

The last argument of the preceding proof demands that we study the shape of proofs (or proof trees) of $M =_{\mathbf{r}} N$ for $M, N \in B$. Since $=_{\mathbf{r}}$ is the reflexive, symmetric, and transitive closure of the one-step reduction $\rightarrow_{\mathbf{r}}$, a calculation to prove $M =_{\mathbf{r}} N$ consists of a series of such one-step reductions in both directions:



In this picture, each expression $L_i \in B$ and each arrow represents a $\rightarrow_{\mathbf{r}}$ relation between two neighboring terms in the sequence. Formally, every $L_i \longrightarrow L_j$ corresponds to $L_i \rightarrow_{\mathbf{r}} L_j$.

The critical insight from this picture is that it might just be possible to reshape such calculations so that all reduction steps go from M to some L and from N to the same L. In other words, if $M =_{\mathbf{r}} N$ perhaps there is an expression L such that $M \twoheadrightarrow_{\mathbf{r}} L$ and $N \twoheadrightarrow_{\mathbf{r}} L$:



If we can prove that such an L always exists for two equal terms, the proof of consistency is finished.

Proof for Theorem 2.1: (remainder) Recall that we have

$$R_1 =_{\mathbf{r}} R_2$$

By the (as yet unproved) claim, there must be an expression L such that

$$R_1 \twoheadrightarrow_{\mathbf{r}} L \qquad \text{and} \qquad R_2 \twoheadrightarrow_{\mathbf{r}} L$$

But elements of R, which are just `t` and `f`, are clearly not reducible to anything except themselves. So $L = R_1$ and $L = R_2$, which means that $R_1 = R_2$.

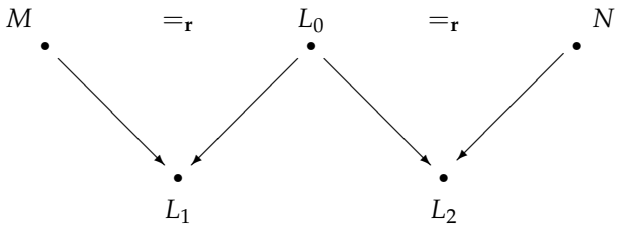
By the preceding reasoning, we have reduced the proof of $eval^{=}_{\mathbf{r}}$'s consistency to a claim about the shape of proof trees that establish $M =_{\mathbf{r}} N$ and our ability to re-shape them. This crucial insight is due to Church and Rosser, who used this idea to analyze the consistency of a language called the λ-calculus (which is the topic of the next chapter). Accordingly the lemma is named after them.[1]

Lemma 2.2 [Consistency for $=_{\mathbf{r}}$]: If $M =_{\mathbf{r}} N$, then there exists an expression L such that $M \twoheadrightarrow_{\mathbf{r}} L$ and $N \twoheadrightarrow_{\mathbf{r}} L$.

Proof for Lemma 2.2: Since we are given the equation $M =_{\mathbf{r}} N$, and since the definition of $=_{\mathbf{r}}$ inductively extends $\twoheadrightarrow_{\mathbf{r}}$ to its reflexive, symmetric, transitive closure, we prove the lemma by induction of the structure of the derivation of $M =_{\mathbf{r}} N$, that is, the structure of its proof tree:

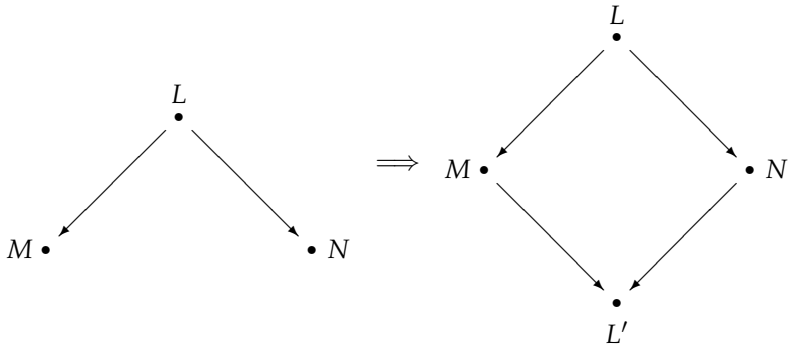
- Base case:
  - **Case** $M \twoheadrightarrow_{\mathbf{r}} N$
    Let $L = N$, and the claim holds.
- Inductive cases:
  - **Case** $M =_{\mathbf{r}} N$ **because** $N =_{\mathbf{r}} M$
    By induction, an L exists for $N =_{\mathbf{r}} M$, and that is the L we want.
  - **Case** $M =_{\mathbf{r}} N$ **because** $M =_{\mathbf{r}} L_0$ **and** $L_0 =_{\mathbf{r}} N$
    By induction, there exists an L_1 such that $M \twoheadrightarrow_{\mathbf{r}} L_1$ and $L_0 \twoheadrightarrow_{\mathbf{r}} L_1$. Also by induction, there exists an L_2 such that $N \twoheadrightarrow_{\mathbf{r}} L_2$ and $L_0 \twoheadrightarrow_{\mathbf{r}} L_2$. In pictures we have:

[1] If we were plain mathematicians studying the λ-calculus, we would call the following lemma a "theorem." We are, however, computer scientists interested in the consistency of our evaluator and thus just use Church and Rosser's idea as an auxiliary step; we do not see it as an end goal of our work.



Now suppose that, whenever an expression L_0 reduces to both L_1 and L_2, there exists some expression L_3 such that $L_1 \twoheadrightarrow_{\mathbf{r}} L_3$ and $L_2 \twoheadrightarrow_{\mathbf{r}} L_3$. Then the claim we want to prove holds, because $M \twoheadrightarrow_{\mathbf{r}} L_3$ and $N \twoheadrightarrow_{\mathbf{r}} L_3$.

Again, we have finished the proof modulo the proof of yet another claim about the reduction system. Specifically, we assumed that if we encounter the situation on the left, it is possible to find a term $L' \in B$ such that we can construct the situation on the right:



This property is called **diamond property** because the picture demands that "reduction branch" can be completed to the shape of a diamond. When the transitive, reflexive, and compatible closure of a notion of reduction, such as $\mathbf{r}$, satisfies the diamond property, it is called **Church-Rosser**.

> **Lemma 2.3 [Church-Rosser ($\twoheadrightarrow_{\mathbf{r}}$)]**: If $L \twoheadrightarrow_{\mathbf{r}} M$ and $L \twoheadrightarrow_{\mathbf{r}} N$, there exists an expression L' such that $M \twoheadrightarrow_{\mathbf{r}} L'$ and $N \twoheadrightarrow_{\mathbf{r}} L'$.

Since $L \twoheadrightarrow_{\mathbf{r}} M$ consists of a series of $\rightarrow_{\mathbf{r}}$ proof steps, it is natural to check whether this latter relation also satisfies a diamond property. If it does, it should be possible to compose the small diamonds for $\rightarrow_{\mathbf{r}}$ to obtain one large diamond for $\twoheadrightarrow_{\mathbf{r}}$. Although the diamond property does not quite hold for $\rightarrow_{\mathbf{r}}$, a sufficiently strong property holds.

Lemma 2.4 [Diamond-like ($\rightarrow_{\mathbf{r}}$)]: If $L \rightarrow_{\mathbf{r}} M$ and $L \rightarrow_{\mathbf{r}} N$ for $M \neq N$, then either

- $M \rightarrow_{\mathbf{r}} N$,
- $N \rightarrow_{\mathbf{r}} M$, or
- there exists an L' such that $M \rightarrow_{\mathbf{r}} L'$ and $N \rightarrow_{\mathbf{r}} L'$.

That is, a "reduction branch" of expressions can be completed to a diamond or there is a "triangle" style completion.

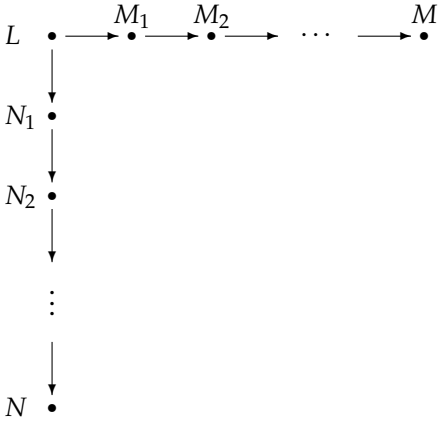
Proof for Lemma 2.4: To prove this lemma, recall that $\rightarrow_{\mathbf{r}}$ is defined inductively as the compatible closure of $\mathbf{r}$. Hence it is natural to assume $L \rightarrow_{\mathbf{r}} M$ and to proceed by structural induction on the structure of its proof:

- Base case:
  - **Case** $L\,\mathbf{r}\,M$
    By the definition of $\mathbf{r}$, there are two subcases.
    - **Case** $L = (\mathtt{f} \bullet B_0)$ **and** $M = B_0$
      The expression L might be reduced in two ways with $\rightarrow_{\mathbf{r}}$ for N: to B_0, or to $(\mathtt{f} \bullet B_0')$ with $B_0 \rightarrow_{\mathbf{r}} B_0'$.
      Since $N \neq M$, $N = (\mathtt{f} \bullet B_0')$ for some B_0' and $B_0\,\mathbf{r}\,B_0'$.
      In that case, take $L' = B_0'$ because $M = B_0 \rightarrow_{\mathbf{r}} B_0'$ and, by the definition of $\mathbf{r}$, $N \rightarrow_{\mathbf{r}} B_0'$.
    - **Case** $L = (\mathtt{t} \bullet B_0)$ **and** $M = \mathtt{t}$
      Given that $M \neq N$, $N = (\mathtt{t} \bullet B_0')$. Therefore, we have $N \rightarrow_{\mathbf{r}} M$, the second possible consequence.
- For the inductive cases, assume without loss of generality that $L \not\mathbf{r} N$; otherwise swap N and M.
  - **Case** $L = (B_1 \bullet B_2)$, $M = (B_1' \bullet B_2)$ **because** $B_1 \rightarrow_{\mathbf{r}} B_1'$
    We have two sub-cases:
    - $N = (B_1'' \bullet B_2)$, where $B_1 \rightarrow_{\mathbf{r}} B_1''$. Since $B_1 \rightarrow_{\mathbf{r}} B_1'$, and $B_1 \rightarrow_{\mathbf{r}} B_1''$, we can apply induction to this branch of the proof tree. If $B_1'' \rightarrow_{\mathbf{r}} B_1'$, then $M \rightarrow_{\mathbf{r}} N$ and the claim holds; similarly, if $B_1' \rightarrow_{\mathbf{r}} B_1''$, then $N \rightarrow_{\mathbf{r}} M$ and the claim holds. Finally, if $B_1' \rightarrow_{\mathbf{r}} B_1'''$ and $B_1'' \rightarrow_{\mathbf{r}} B_1'''$, then $M \rightarrow_{\mathbf{r}} (B_1''' \bullet B_2)$ and $N \rightarrow_{\mathbf{r}} (B_1''' \bullet B_2)$, and the claim holds with $L' = (B_1''' \bullet B_2)$.
    - $N = (B_1 \bullet B_2')$ with $B_2 \rightarrow_{\mathbf{r}} B_2'$. Since $B_1 \rightarrow_{\mathbf{r}} B_1'$, $N \rightarrow_{\mathbf{r}} (B_1' \bullet B_2')$. Similarly, $M \rightarrow_{\mathbf{r}} (B_1' \bullet B_2')$. Thus, the claim holds with $L' = (B_1' \bullet B_2')$.

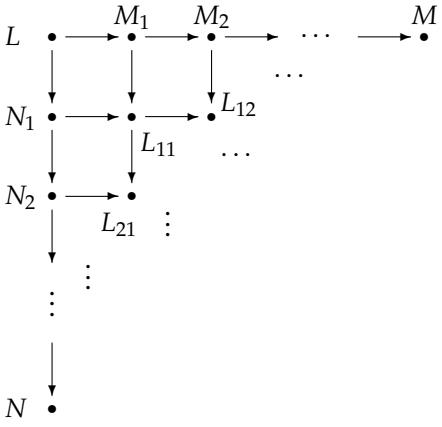
– **Case** $L = (B_1 \bullet B_2)$, $M = (B_1 \bullet B_2')$ **because** $B_2 \to_{\mathbf{r}} B_2'$
This case is analogous to the previous one.

Now that we know that the one-step reduction satisfies a diamond-like property, we can show that its transitive–reflexive closure satifies the diamond property. That is, we can prove the "big diamond" lemma.

Proof for Lemma 2.3: Assume that $L \twoheadrightarrow_{\mathbf{r}} M$ and $L \twoheadrightarrow_{\mathbf{r}} N$. By the inductive definition of the reduction relation $\twoheadrightarrow_{\mathbf{r}}$, $L \to_{\mathbf{r}}^{m} M$ and $L \to_{\mathbf{r}}^{n} N$ for some $m, n \in \mathbb{N}$, where $\to_{\mathbf{r}}^{m}$ means m steps with $\to_{\mathbf{r}}$. Pictorially, we have



Using the diamond-like property for the one-step reduction, we can now fill in expressions L_{11}, L_{21}, L_{12}, etc. until the entire square is filled out:



Formally, this proof can also be cast as an induction on the number of reduction steps.

Exercise 2.1. Prove lemma 2.3 formally.

The preceding arguments also show that $M =_{\mathbf{r}} R$ if and only if $M \twoheadrightarrow_{\mathbf{r}} R$. Consequently, the theorem-answer for our second question—whether the two evaluators are equal—falls out as a consequence.

Theorem 2.5: $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}} = eval_{\mathbf{r}}^{=_{\mathbf{r}}}$

Proof for Theorem 2.5: We show that each element of $eval_{\mathbf{r}}^{=_{\mathbf{r}}}$ is also in $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}$ and vice versa. Thus, assume that $eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_0) = R_0$. By definition, $B_0 =_{\mathbf{r}} R_0$. From lemma 2.2, $B_0 \twoheadrightarrow_{\mathbf{r}} R_0$ because the latter is not reducible. This, of course, implies $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}(B_0) = R_0$ according to the definition of this second evaluator.

For the other direction, assume $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}(B_1) = R_1$, which means $B_1 \twoheadrightarrow_{\mathbf{r}} R_1$ by the definition of the function. In turn, the definition of $=_{\mathbf{r}}$ as symmetric (and equivalence) closure of $\twoheadrightarrow_{\mathbf{r}}$ implies $eval_{\mathbf{r}}^{=_{\mathbf{r}}}(B_1) = R_1$.

For the programmer and anyone else who wishes to reason about the evaluation process of B expressions, the theorem says that symmetric reasoning steps do not help in the evaluation of B expressions. The directional $\twoheadrightarrow_{\mathbf{r}}$ relation suffices.[2]

At this point, we know that the evaluators are functions, and that the two evaluators are identical. The only question left is whether the evaluators terminate for every program that they are given.

Theorem 2.6: For all B_0, there is an R_0 such that $eval_{\mathbf{r}}(B_0) = R_0$.

While $eval_{\mathbf{r}}(B_0) = R_0$ does not specify which evaluator is meant, it is clear that the theorem refers to both.

Proof for Theorem 2.6: Consider a single reduction step:

$$B_0 \rightarrow_{\mathbf{r}} B_1$$

Because of the definition of $\rightarrow_{\mathbf{r}}$ as the compatible closure of $\mathbf{r}$, this means that B_1 contains fewer instances of $\bullet$ than B_0. Since

[2]The next few chapters introduce languages for which such apparent backward steps truly shorten calculations on occasion.

B_0 contains a finite number k of $\bullet$ and since it is impossible for an element of B to contain fewer than zero instances of $\bullet$, a chain of the shape

$$B_0 \rightarrow_{\mathbf{r}} B_1 \rightarrow_{\mathbf{r}} \ldots \rightarrow_{\mathbf{r}} B_n$$

may contain at most $n \leq k$ number of steps.

Thus, the question is whether a non-result expression from B is always reducible. Put differently, we are asking whether there is a B_j for every B_i not in R such that $B_i \rightarrow_{\mathbf{r}} B_j$. To prove this claim, we proceed by induction on the structure of the expression B_i:

- **Case** $B_i = \mathtt{t}$
  This case contradicts the assumption that B_i is not in R.
- **Case** $B_i = \mathtt{f}$
  As for the first case, B_i is not in R.
- **Case** $B_i = B_1 \bullet B_2$
  If B_1 is in R, the claim is established because $\mathbf{r}$ reduces B_i to either $\mathtt{t}$ or B_2 and either of which serves as B_j.

  Otherwise, by induction, B_1 is reducible to B_1' and, by construction, $B_1 \bullet B_2 \rightarrow_{\mathbf{r}} B_1' \bullet B_2$, and we take this last expression as B_j.

Using this theorem, it is possible to decide the equality of B expressions. The algorithm simply evaluates both expressions and compares the results. Of course, full-fledged programming languages include arbitrary non-terminating expressions and thus preclude a programmer from deciding the equivalence of expressions in this way.