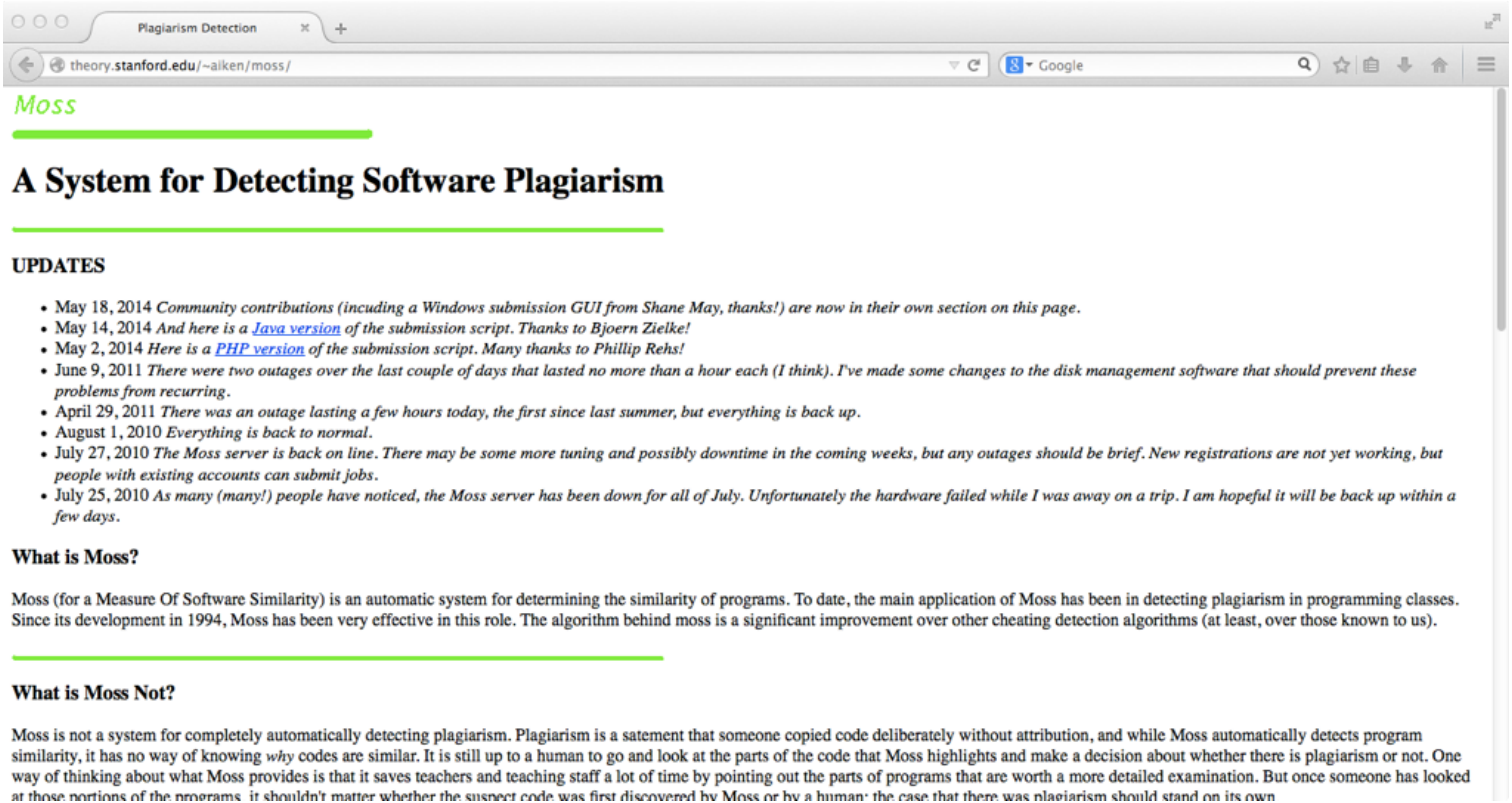
Code-Specific, Sensitive, and Configurable Plagiarism Detection

Kyle Dewey, Ben Hardekopf

Simple(?) Problem

- Want to perform automated plagiarism detection of Scala code originating from class assignments
- Doing this by hand is time-consuming, tedious, and error-prone

MOSS



Moss

A System for Detecting Software Plagiarism

UPDATES

- May 18, 2014 *Community contributions (incuding a Windows submission GUI from Shane May, thanks!) are now in their own section on this page.*
- May 14, 2014 *And here is a Java version of the submission script. Thanks to Bjoern Zielke!*
- May 2, 2014 *Here is a PHP version of the submission script. Many thanks to Phillip Rehs!*
- June 9, 2011 *There were two outages over the last couple of days that lasted no more than a hour each (I think). I've made some changes to the disk management software that should prevent these problems from recurring.*
- April 29, 2011 *There was an outage lasting a few hours today, the first since last summer, but everything is back up.*
- August 1, 2010 *Everything is back to normal.*
- July 27, 2010 *The Moss server is back on line. There may be some more tuning and possibly downtime in the coming weeks, but any outages should be brief. New registrations are not yet working, but people with existing accounts can submit jobs.*
- July 25, 2010 *As many (many!) people have noticed, the Moss server has been down for all of July. Unfortunately the hardware failed while I was away on a trip. I am hopeful it will be back up within a few days.*

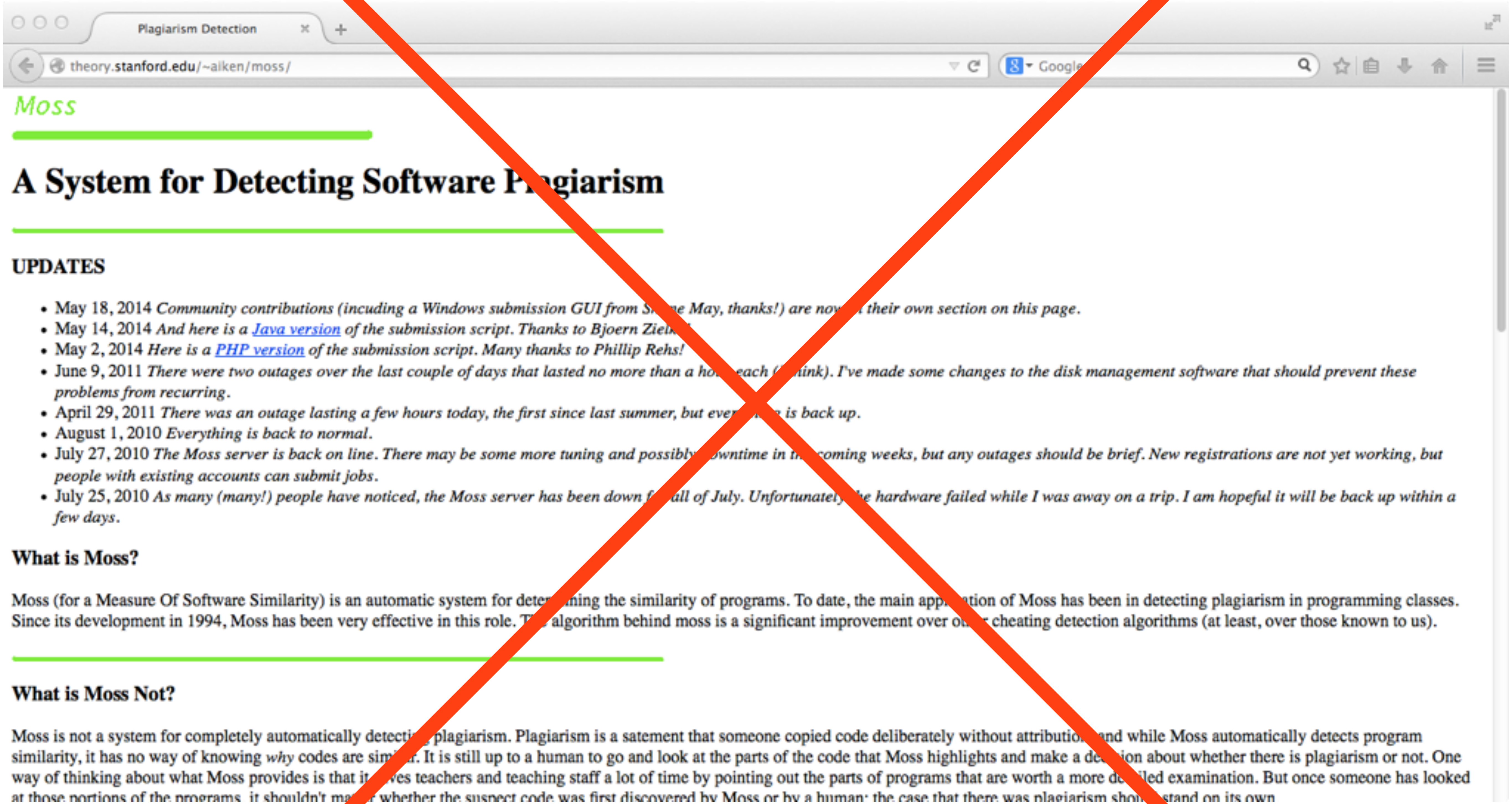
What is Moss?

Moss (for a Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Since its development in 1994, Moss has been very effective in this role. The algorithm behind moss is a significant improvement over other cheating detection algorithms (at least, over those known to us).

What is Moss Not?

Moss is not a system for completely automatically detecting plagiarism. Plagiarism is a satement that someone copied code deliberately without attribution, and while Moss automatically detects program similarity, it has no way of knowing *why* codes are similar. It is still up to a human to go and look at the parts of the code that Moss highlights and make a decision about whether there is plagiarism or not. One way of thinking about what Moss provides is that it saves teachers and teaching staff a lot of time by pointing out the parts of programs that are worth a more detailed examination. But once someone has looked at those portions of the programs, it shouldn't matter whether the suspect code was first discovered by Moss or by a human; the case that there was plagiarism should stand on its own

MOSS



Moss

A System for Detecting Software Plagiarism

UPDATES

- May 18, 2014 *Community contributions (incuding a Windows submission GUI from Shane May, thanks!) are now in their own section on this page.*
- May 14, 2014 *And here is a [Java version](#) of the submission script. Thanks to Bjoern Zielke!*
- May 2, 2014 *Here is a [PHP version](#) of the submission script. Many thanks to Phillip Rehs!*
- June 9, 2011 *There were two outages over the last couple of days that lasted no more than a hour each (I think). I've made some changes to the disk management software that should prevent these problems from recurring.*
- April 29, 2011 *There was an outage lasting a few hours today, the first since last summer, but everything is back up.*
- August 1, 2010 *Everything is back to normal.*
- July 27, 2010 *The Moss server is back on line. There may be some more tuning and possibly downtime in the coming weeks, but any outages should be brief. New registrations are not yet working, but people with existing accounts can submit jobs.*
- July 25, 2010 *As many (many!) people have noticed, the Moss server has been down for all of July. Unfortunately the hardware failed while I was away on a trip. I am hopeful it will be back up within a few days.*

What is Moss?

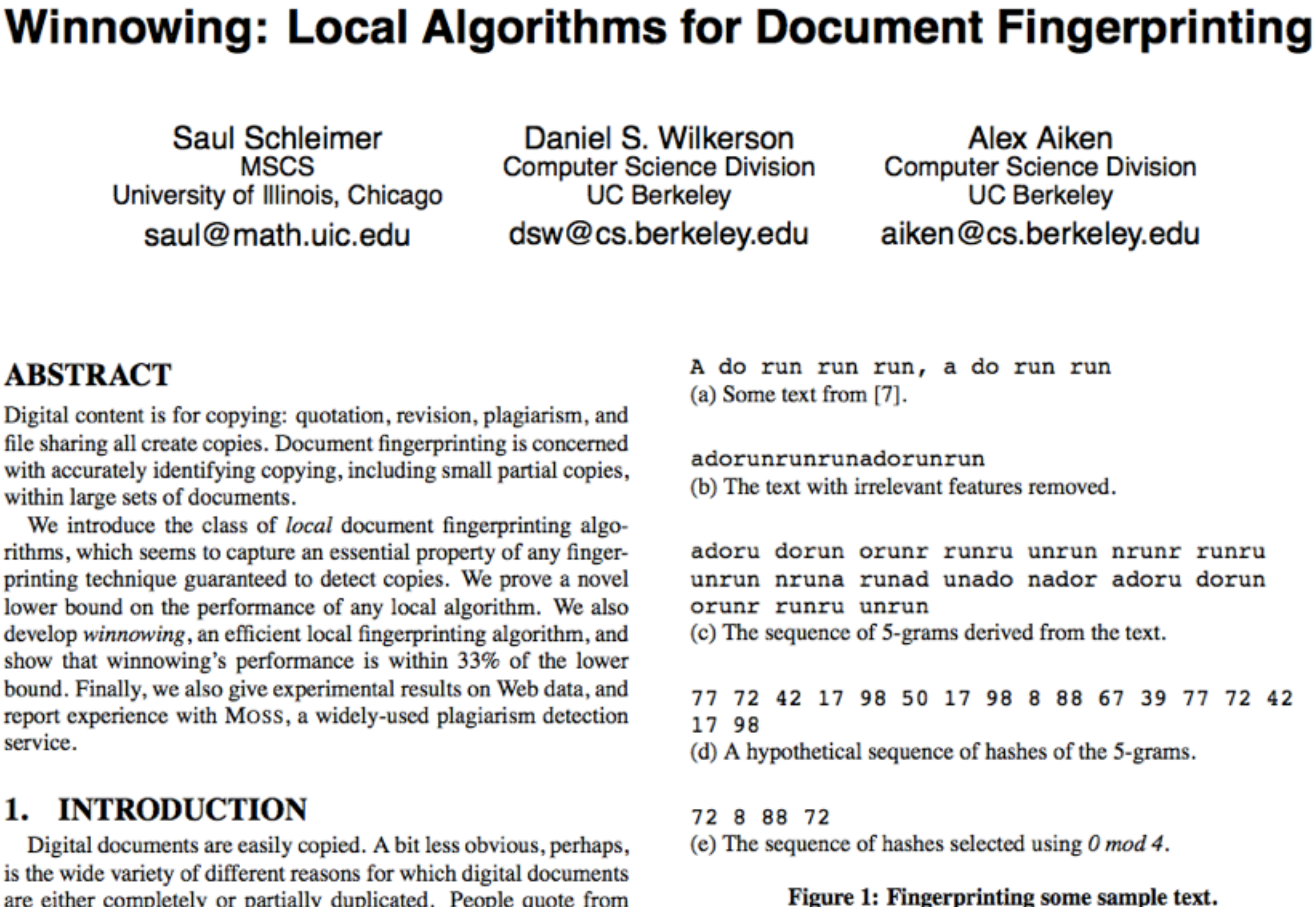
Moss (for a Measure Of Software Similarity) is an automatic system for determining the similarity of programs. To date, the main application of Moss has been in detecting plagiarism in programming classes. Since its development in 1994, Moss has been very effective in this role. The algorithm behind moss is a significant improvement over other cheating detection algorithms (at least, over those known to us).

What is Moss Not?

Moss is not a system for completely automatically detecting plagiarism. Plagiarism is a satement that someone copied code deliberately without attribution, and while Moss automatically detects program similarity, it has no way of knowing *why* codes are similar. It is still up to a human to go and look at the parts of the code that Moss highlights and make a decision about whether there is plagiarism or not. One way of thinking about what Moss provides is that it saves teachers and teaching staff a lot of time by pointing out the parts of programs that are worth a more detailed examination. But once someone has looked at those portions of the programs, it shouldn't matter whether the suspect code was first discovered by Moss or by a human; the case that there was plagiarism should stand on its own

No Scala Support, Proprietary

MOSS



Winnowing: Local Algorithms for Document Fingerprinting

Saul Schleimer
MSCS
University of Illinois, Chicago
saul@math.uic.edu

Daniel S. Wilkerson
Computer Science Division
UC Berkeley
dsw@cs.berkeley.edu

Alex Aiken
Computer Science Division
UC Berkeley
aiken@cs.berkeley.edu

ABSTRACT

Digital content is for copying: quotation, revision, plagiarism, and file sharing all create copies. Document fingerprinting is concerned with accurately identifying copying, including small partial copies, within large sets of documents.

We introduce the class of *local* document fingerprinting algorithms, which seems to capture an essential property of any fingerprinting technique guaranteed to detect copies. We prove a novel lower bound on the performance of any local algorithm. We also develop *winnowing*, an efficient local fingerprinting algorithm, and show that winnowing's performance is within 33% of the lower bound. Finally, we also give experimental results on Web data, and report experience with MOSS, a widely-used plagiarism detection service.

1. INTRODUCTION

Digital documents are easily copied. A bit less obvious, perhaps, is the wide variety of different reasons for which digital documents are either completely or partially duplicated. People quote from

```
A do run run run, a do run run
```

(a) Some text from [7].

```
adorunrunrunadorunrun
```

(b) The text with irrelevant features removed.

```
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
```

(c) The sequence of 5-grams derived from the text.

```
77 72 42 17 98 50 17 98 8 88 67 39 77 72 42
17 98
```

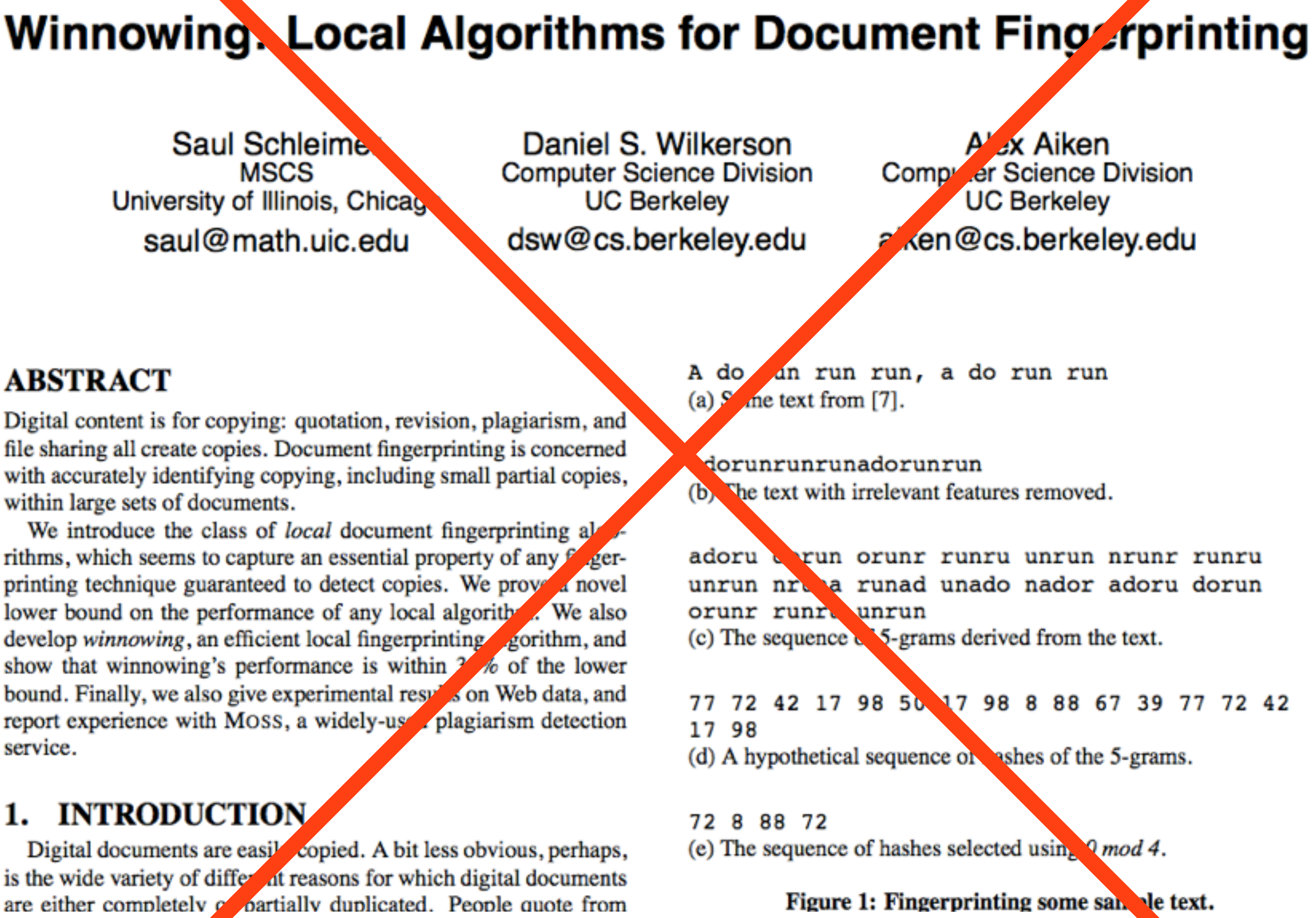
(d) A hypothetical sequence of hashes of the 5-grams.

```
72 8 88 72
```

(e) The sequence of hashes selected using *0 mod 4*.

Figure 1: Fingerprinting some sample text.

MOSS



Winnowing: Local Algorithms for Document Fingerprinting

Saul Schleimer
MSCS
University of Illinois, Chicago
saul@math.uic.edu

Daniel S. Wilkerson
Computer Science Division
UC Berkeley
dsw@cs.berkeley.edu

Alex Aiken
Computer Science Division
UC Berkeley
aiken@cs.berkeley.edu

ABSTRACT

Digital content is for copying: quotation, revision, plagiarism, and file sharing all create copies. Document fingerprinting is concerned with accurately identifying copying, including small partial copies, within large sets of documents.

We introduce the class of *local* document fingerprinting algorithms, which seems to capture an essential property of any fingerprinting technique guaranteed to detect copies. We prove a novel lower bound on the performance of any local algorithm. We also develop *winnowing*, an efficient local fingerprinting algorithm, and show that winnowing's performance is within 33% of the lower bound. Finally, we also give experimental results on Web data, and report experience with Moss, a widely-used plagiarism detection service.

1. INTRODUCTION

Digital documents are easily copied. A bit less obvious, perhaps, is the wide variety of different reasons for which digital documents are either completely or partially duplicated. People quote from

```
A do run run run, a do run run
```

(a) Some text from [7].

```
adorunrunrunadorunrun
```

(b) The text with irrelevant features removed.

```
adoru dorun orunr runru unrun nrunr runru
unrun nruna runad unado nador adoru dorun
orunr runru unrun
```

(c) The sequence of 5-grams derived from the text.

```
77 72 42 17 98 50 17 98 8 88 67 39 77 72 42
17 98
```

(d) A hypothetical sequence of hashes of the 5-grams.

```
72 8 88 72
```

(e) The sequence of hashes selected using *0 mod 4*.

Figure 1: Fingerprinting some sample text.

Lackluster Results

Fundamental Problems

- Existing plagiarism detection tools tend to suffer from at least one of the following problems:
  - Meant for plaintext (loses code-specific information)
  - Lossy (loses information in general)
  - Difficult to specialize for individual assignments (lack of configurability)

A Solution Must:

- Be aware of syntax
- Use all available information
- Allow for easy, highly specific configurability

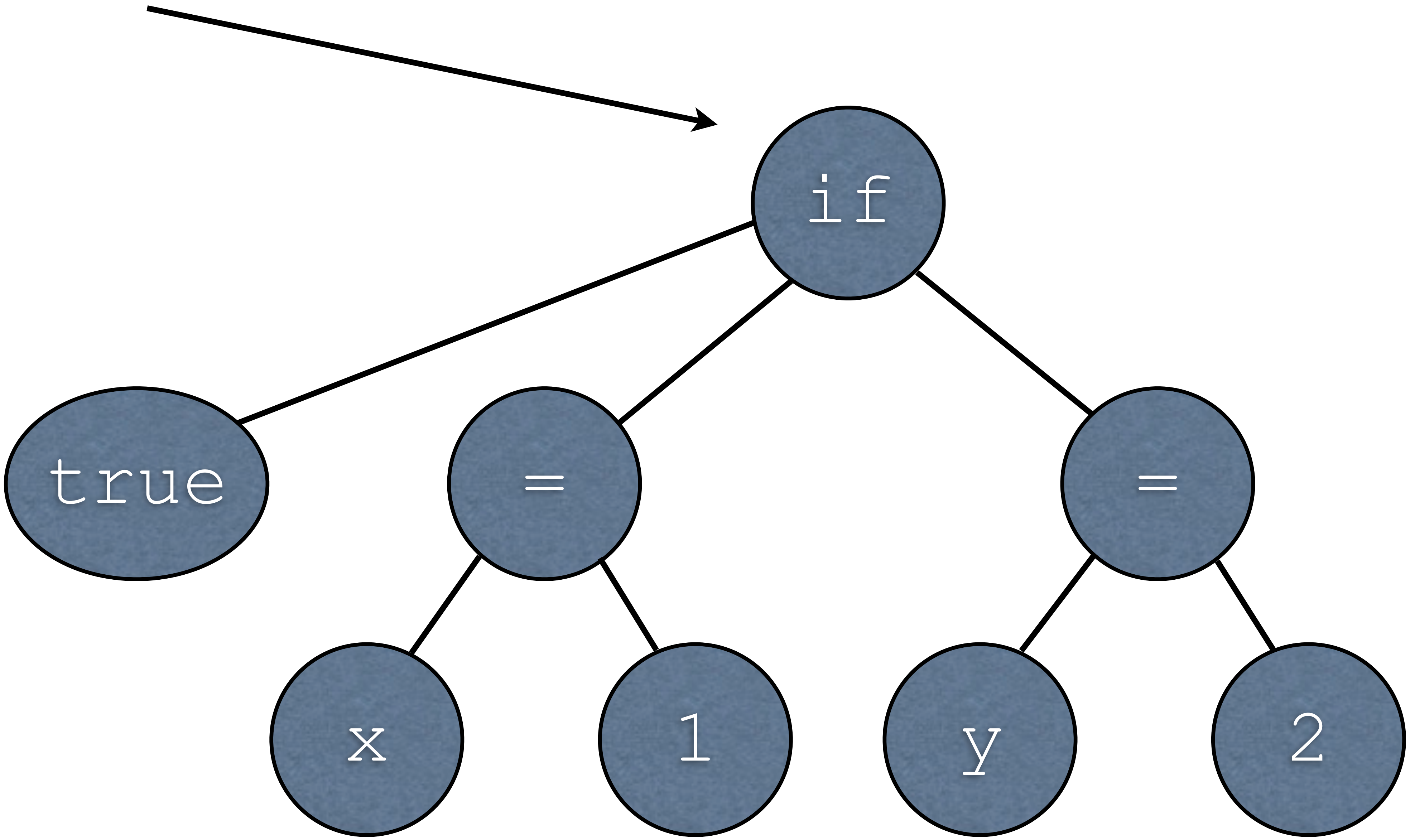
Our Solution

Step 1: Parse in Code

```
if (true) {
  x = 1;
} else {
  y = 2;
}
```

Step 1: Parse in Code

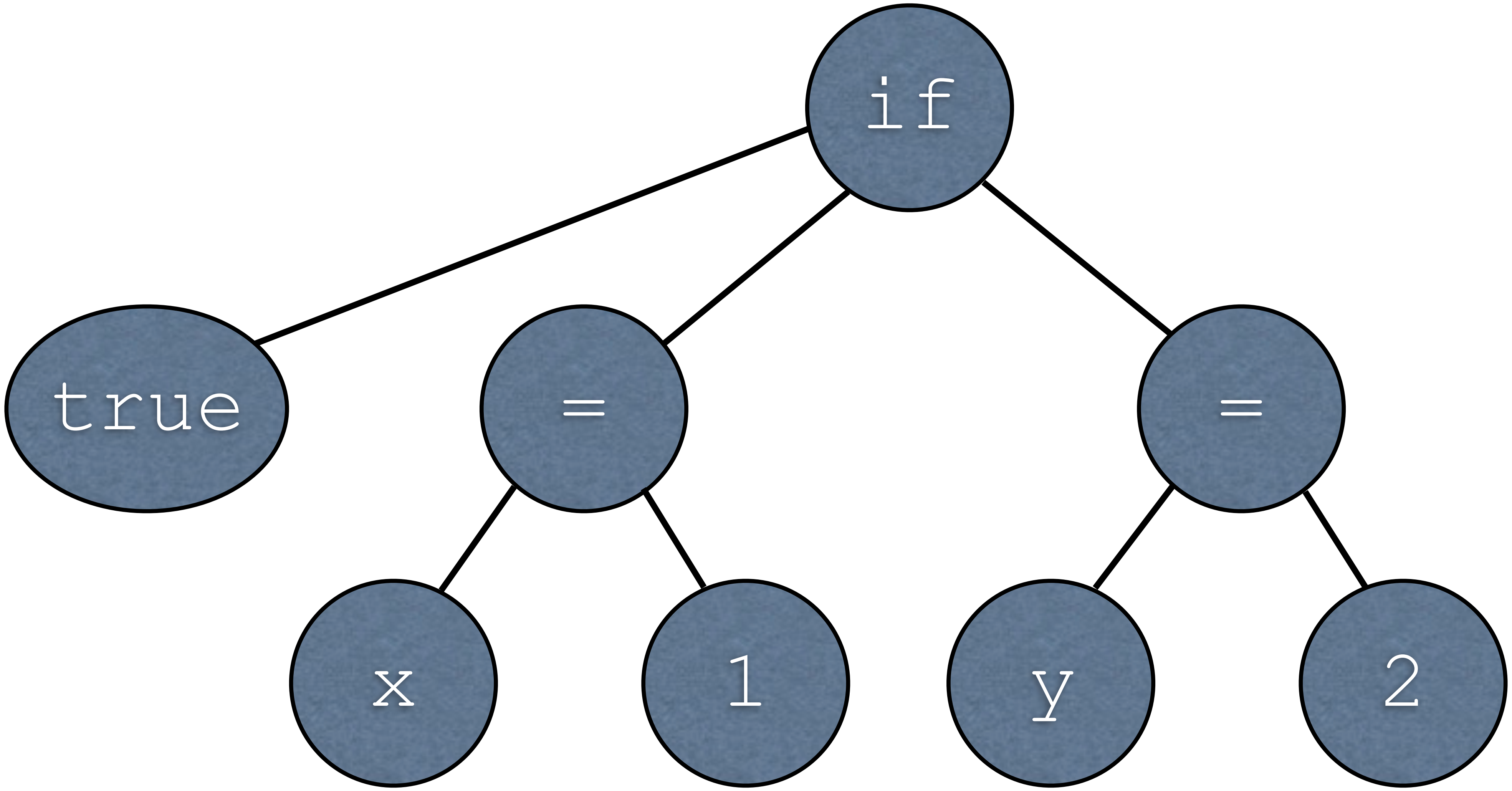
```
if (true) {
  x = 1;
} else {
  y = 2;
}
```



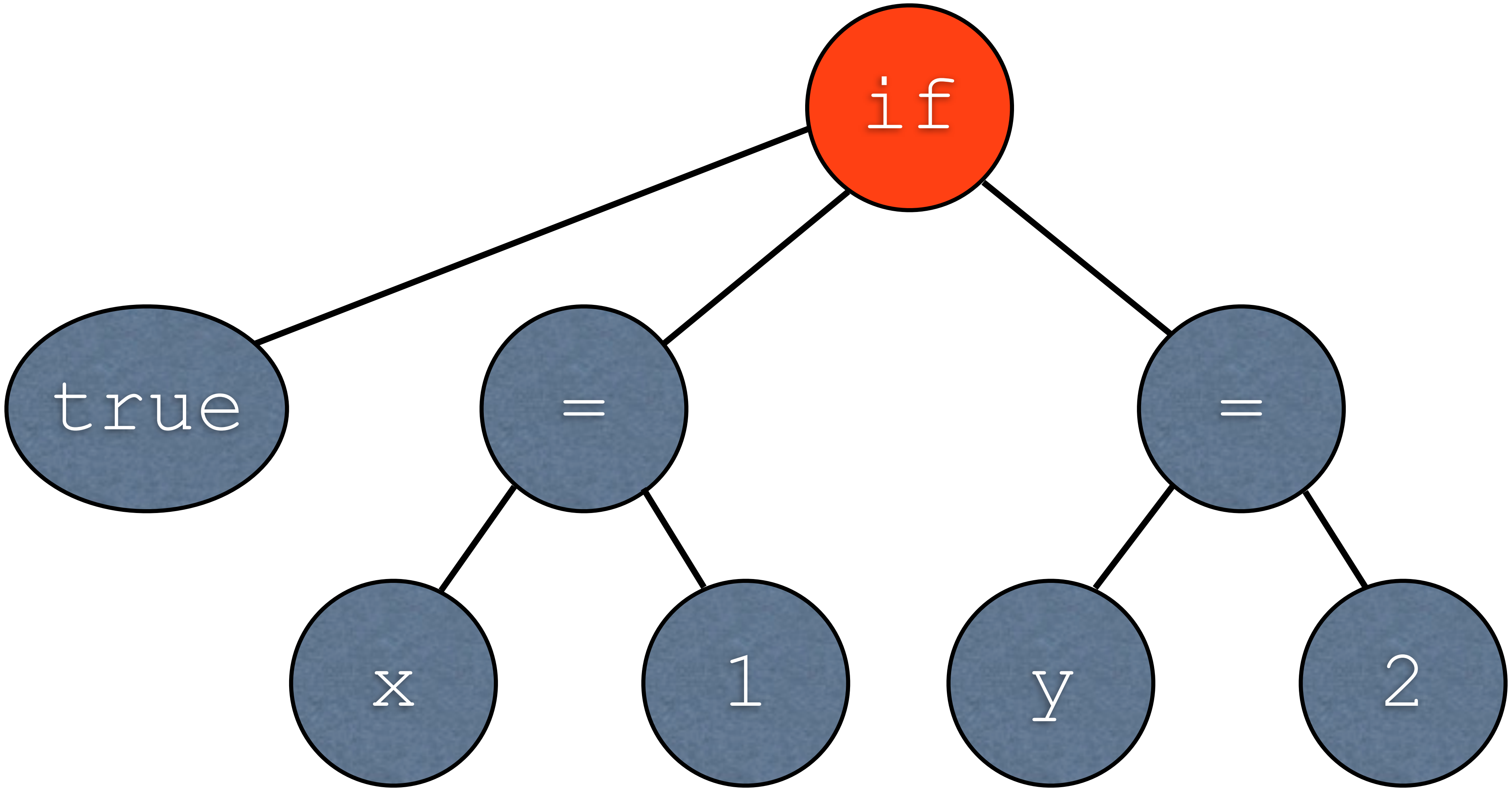
A Solution Must:

- Be aware of syntax
- Use all available information
- Allow for easy, highly specific configurability

Step 2: Extract Features via a User-Defined Function

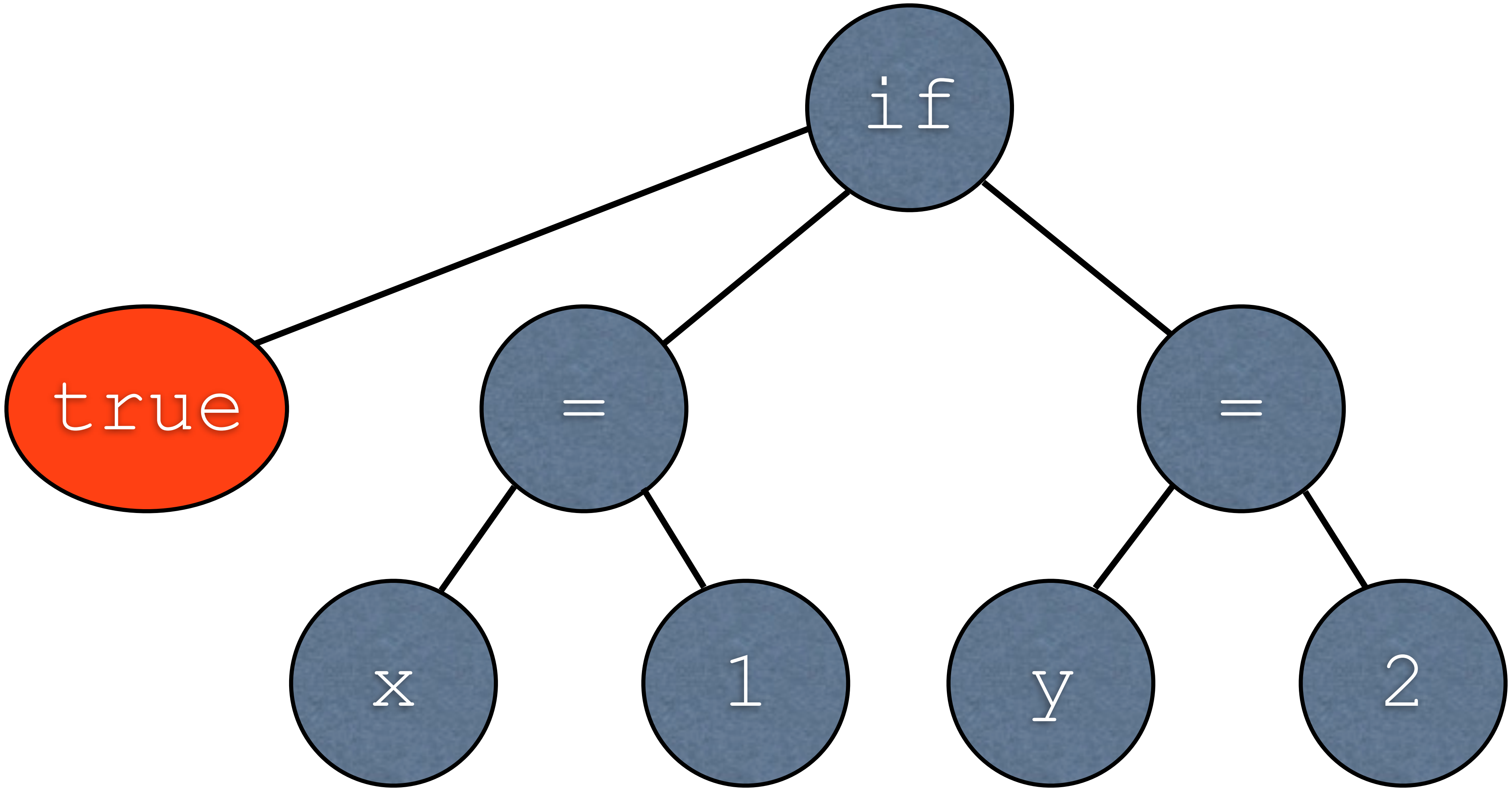


Step 2: Extract Features via a User-Defined Function



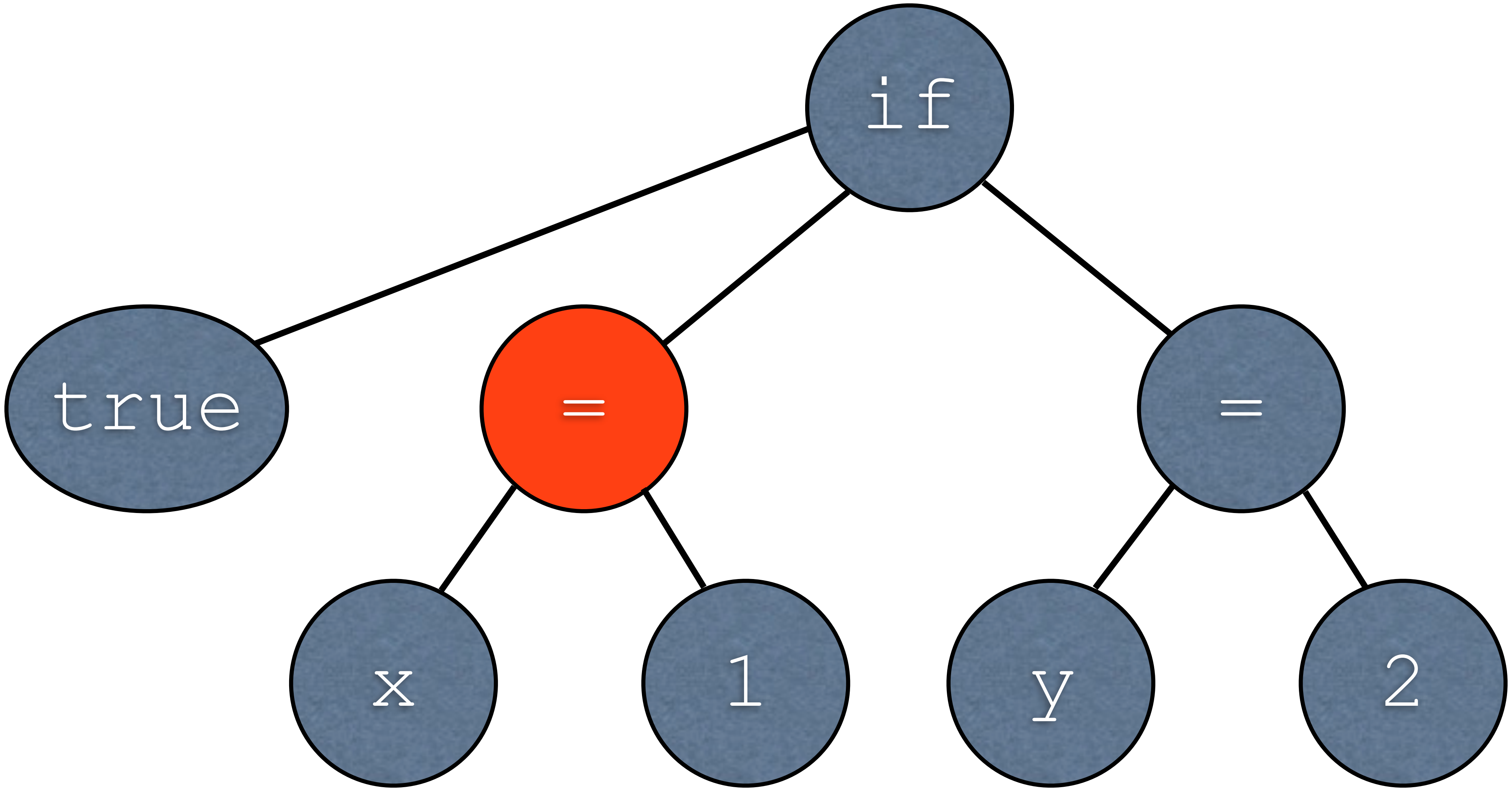
if

Step 2: Extract Features via a User-Defined Function



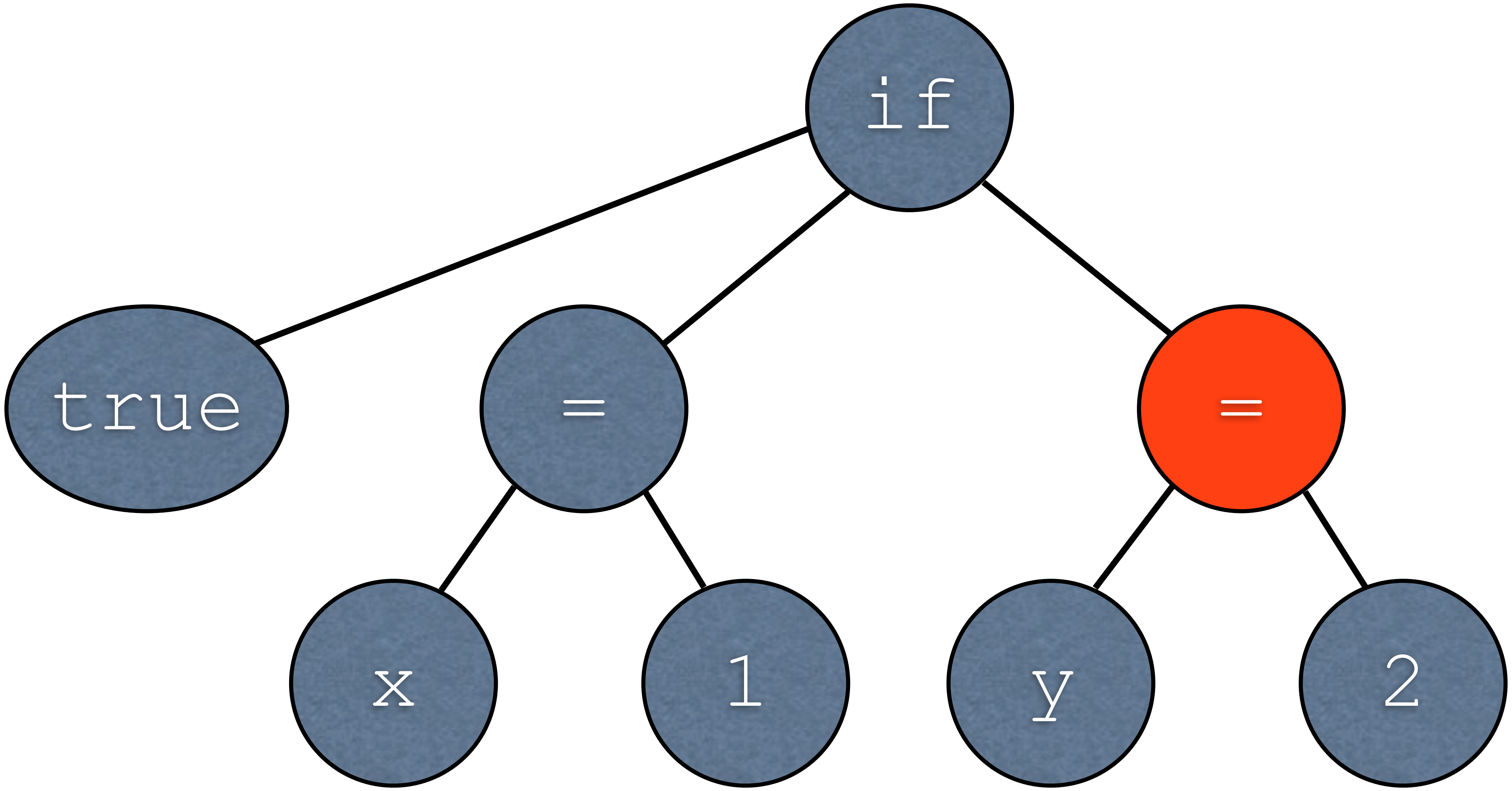
if

Step 2: Extract Features via a User-Defined Function



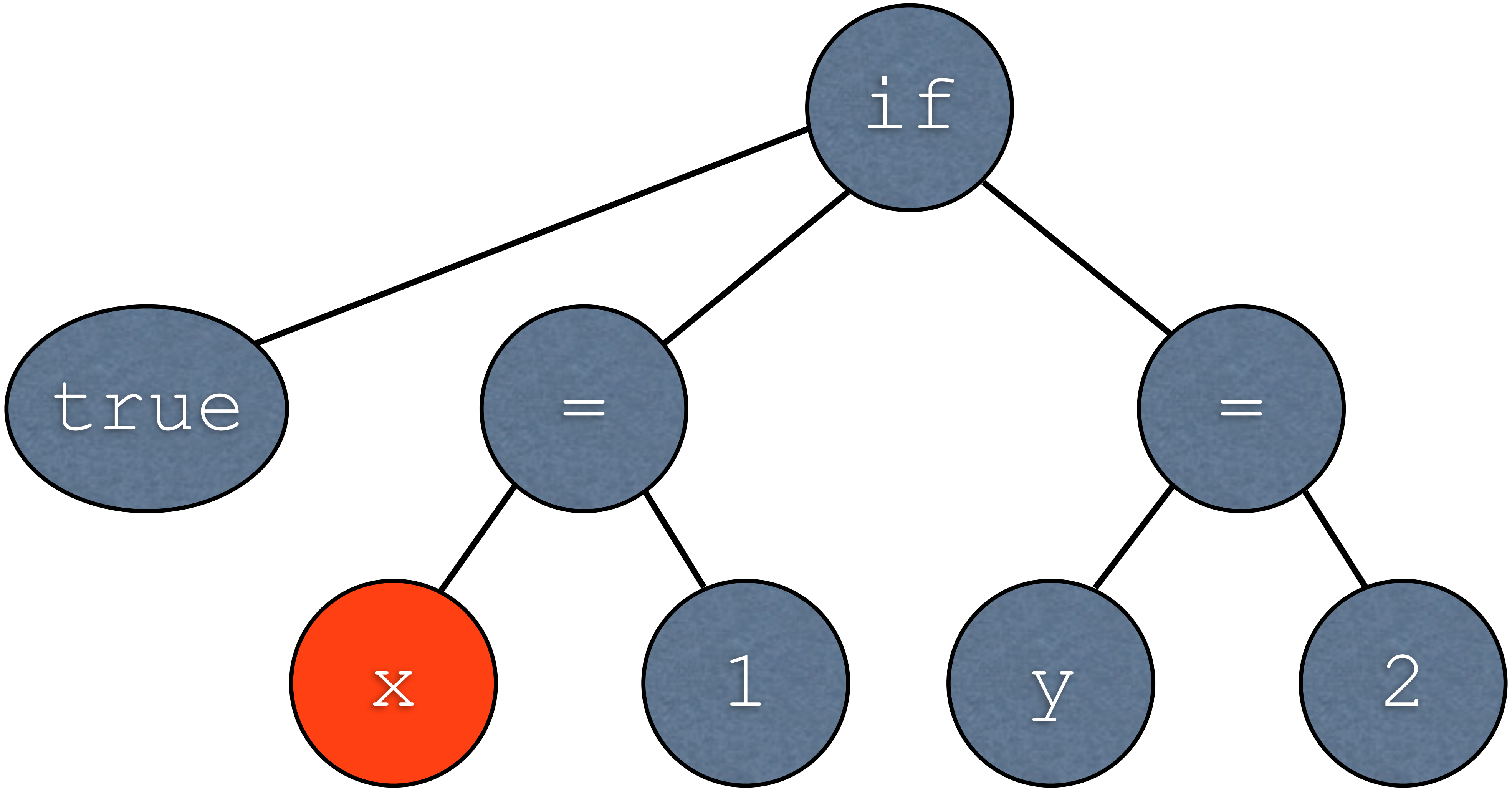
if	eq

Step 2: Extract Features via a User-Defined Function



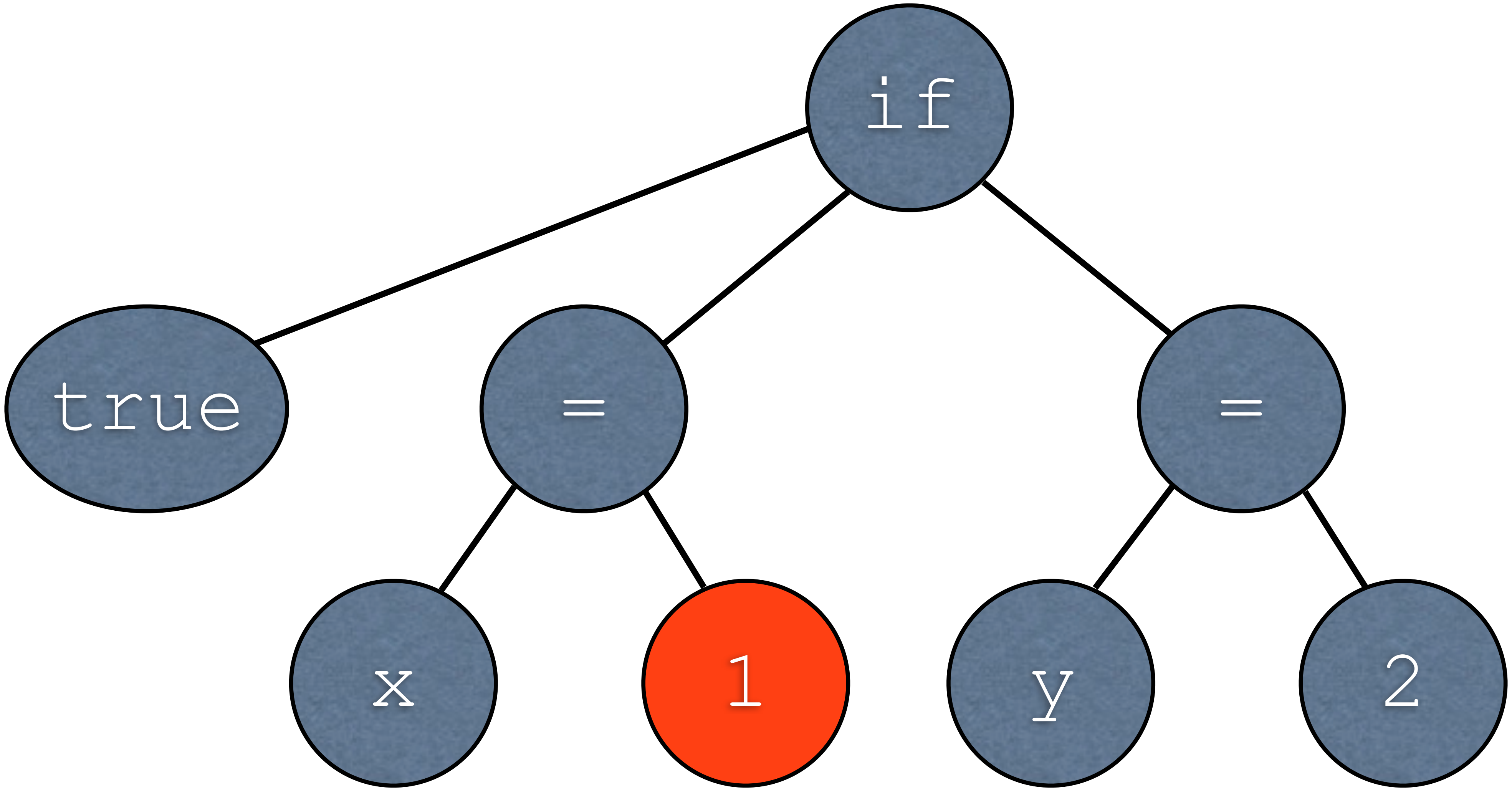
if	eq	eq

Step 2: Extract Features via a User-Defined Function



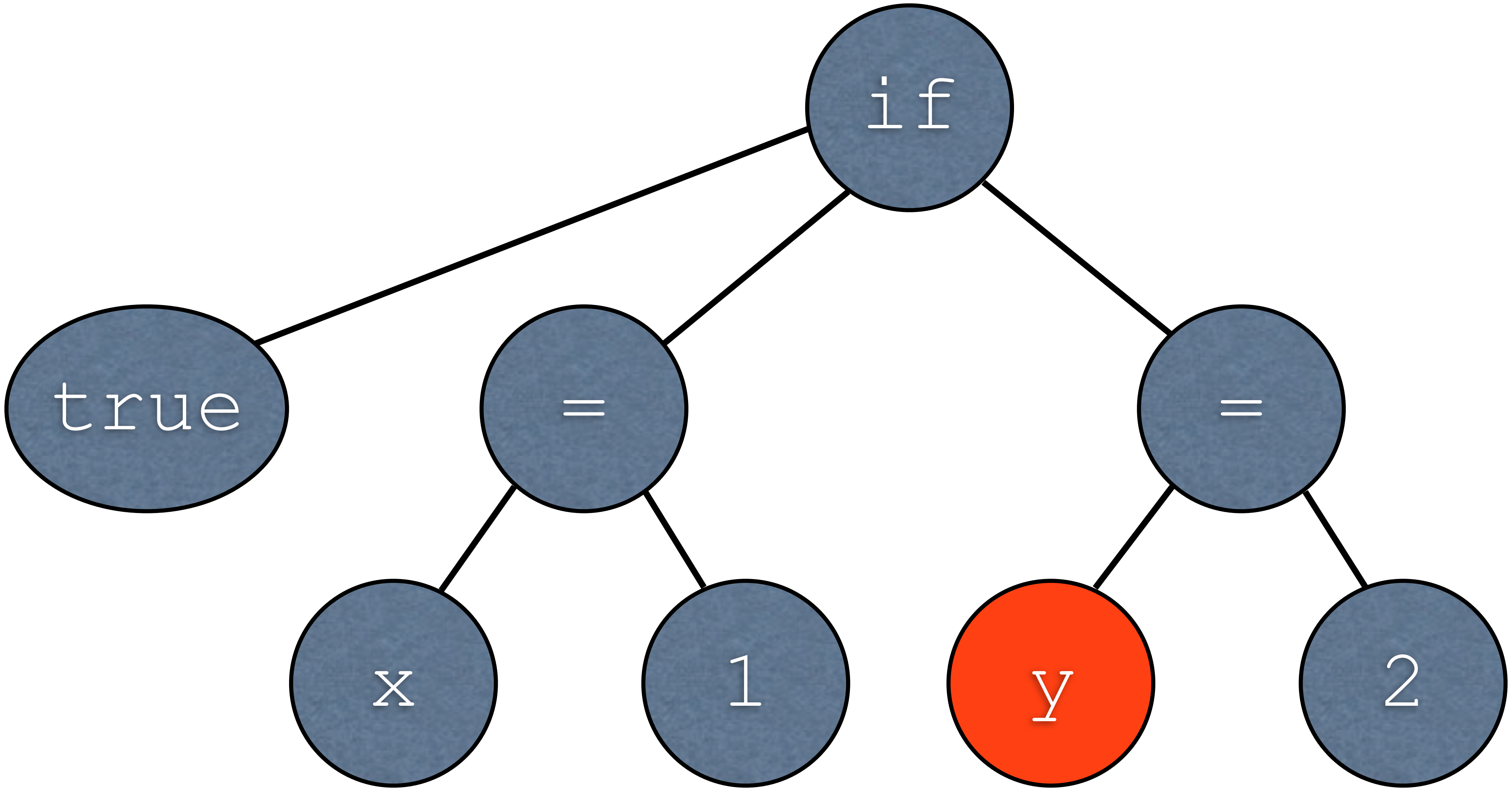
if	eq	eq	var

Step 2: Extract Features via a User-Defined Function



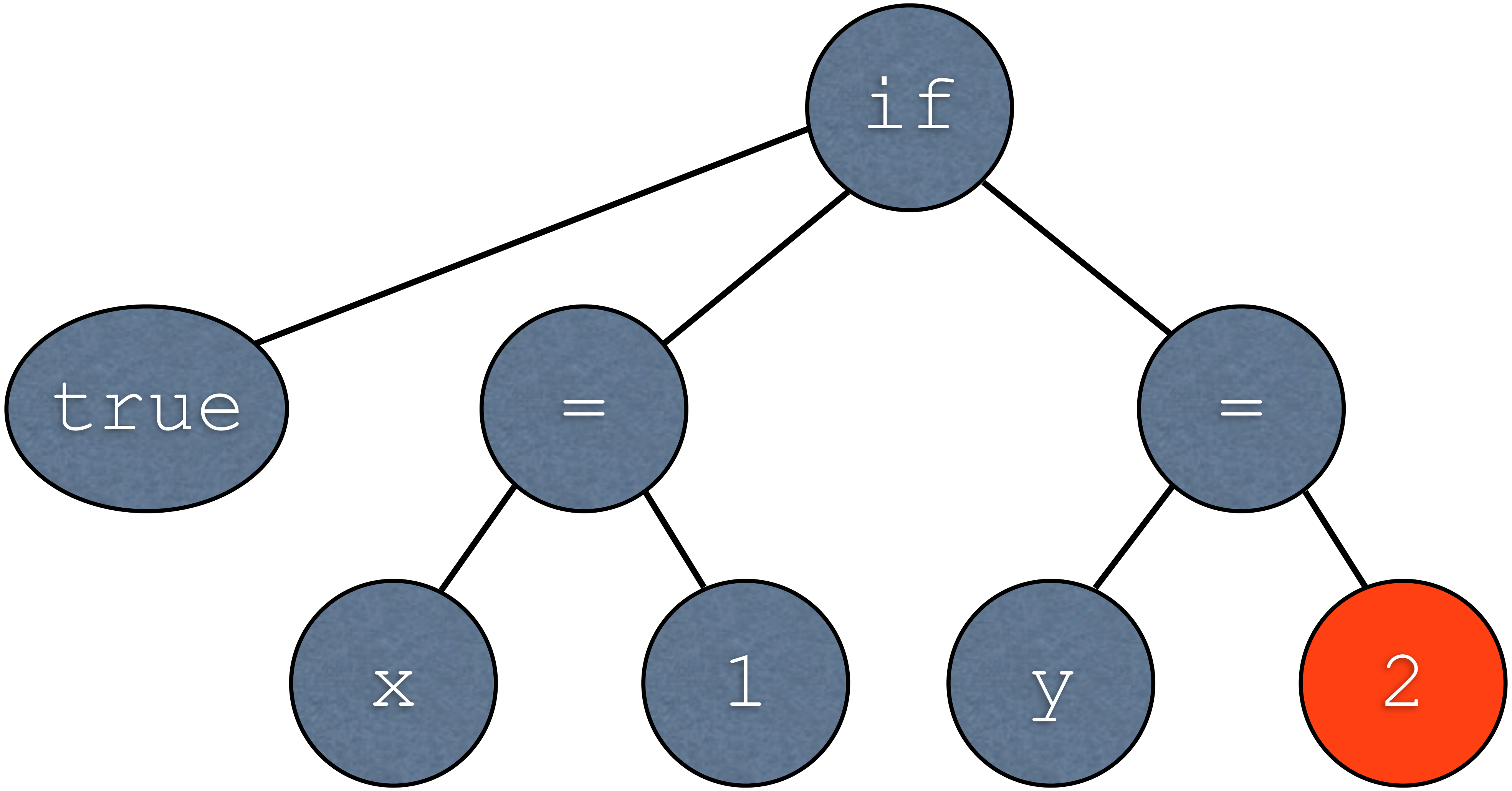
if	eq	eq	var

Step 2: Extract Features via a User-Defined Function



if	eq	eq	var	var

Step 2: Extract Features via a User-Defined Function



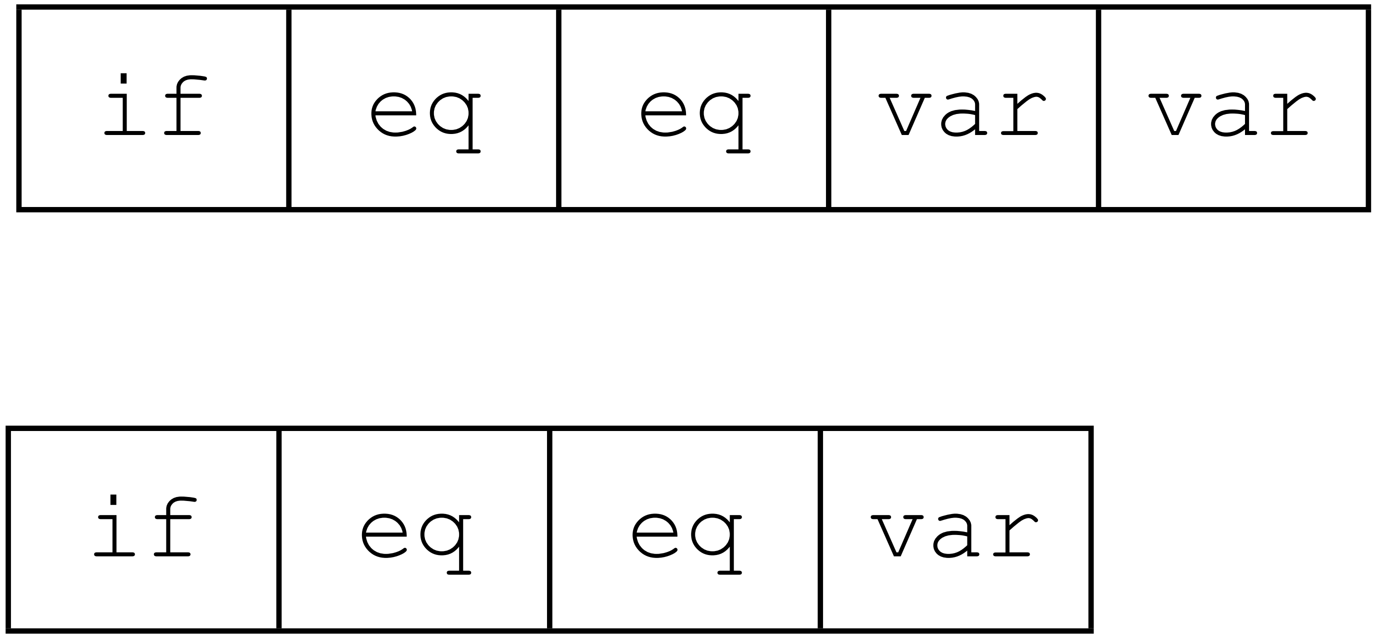
if	eq	eq	var	var

A Solution Must:

- Be aware of syntax
- Use all available information
- Allow for easy, highly specific configurability

Step 3: Align Using Optimal Sequence Alignment Algorithm

Takes two feature vectors (from two separate programs)...



if	eq	eq	var	var

if	eq	eq	var

Step 3: Align Using Optimal Sequence Alignment Algorithm

...along with a scoring function for comparing two features.

```
int score(Feature a,
          Feature b) {
  if (a == b) {
    return 1;
  } else {
    return -1;
  }
}
```

Step 3: Align Using Optimal Sequence Alignment Algorithm

Returns an optimal alignment and a numeric score for the alignment

`if`	`eq`	`eq`	`var`	`var`
`if`	`eq`	`eq`	`---`	`var`

```
Score: 9
```

A Solution Must:

- Be aware of syntax
- Use all available information (optimal)
- Allow for easy, highly specific configurability (scoring function)

Key Differences from Related Work

- We consider whole abstract syntax trees, not just tokens
- User-defined feature extraction
- User-defined scoring

Application of our Technique to Scala

Components to Plug In

- Feature extractor
- Pairwise feature scoring function

Feature Extraction Phase One

- Extract out methods and sort by size
    - Tolerant of reordering
- Process method-by-method, forming a single feature sequence

Feature Extraction Phase Two

- Do a traversal over each method, emitting feature information for forms related to control flow (e.g., `if`) and variable binding
  - Tend to be highly unique to a solution
  - Literals and names are put into equivalence classes (e.g, all literals have the same feature)

Scoring Function

- Quite naive: +2 for any two matched features, and -1 for and mismatches
- Gaps (the −−− part shown before) uniformly have a -1 score
- Could be much more complex if so desired

Putting it All Together

- Implemented via a `scalac` compiler plugin, which gives direct access to the parser
- Series of scripts on top for running over multiple pairs of code

Evaluation and Results

- Applied to a previous assignment which had been manually annotated for plagiarism
- All known cases of plagiarism were high-scoring
- Only one unannotated high-scoring case (which turned out to have been missed)
- Remainder were low-scoring
- Took only a few minutes

Future Work

- Also have prototype for Prolog, which has proven more difficult to get right
    - Syntax is so simple that it provides very little information about control flow
    - Dynamically typed so less information available syntactically at all
    - Currently, lots of false positives