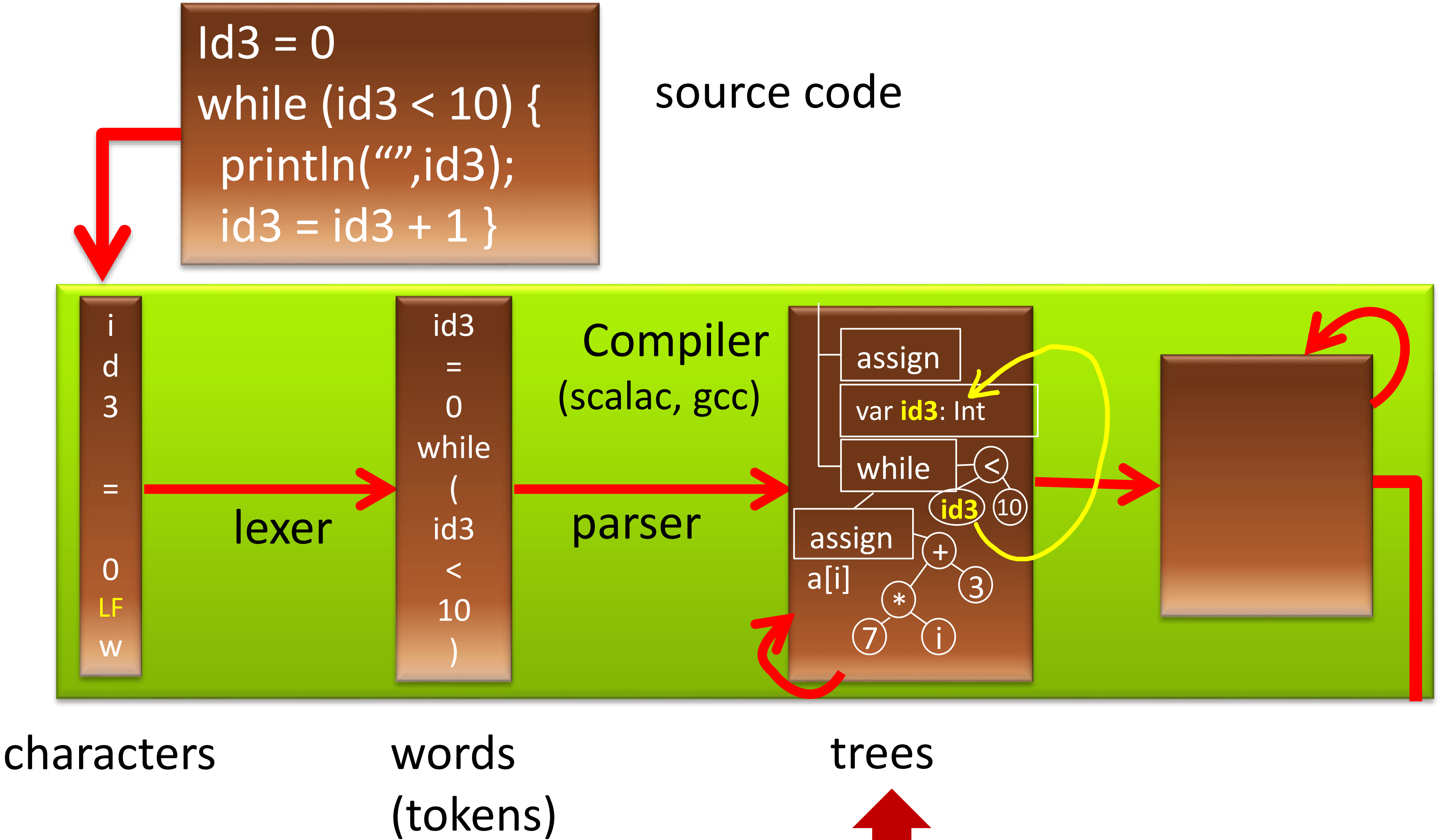


Name Analysis:

making sense of trees;
converting them into **graphs**:
connect **uses** and **declarations**

Errors Detected So Far

- File input: file does not exist
- Lexer: unknown token, string not closed before end of file, …
- Parser: syntax error - unexpected token, cannot parse given non-terminal
- Name analyzer: unknown variable, …
- Type analyzer: applying function to argument of wrong type, …
- Data-flow analyzer: variable read before being written, …

Showing Good Errors with Syntax Trees

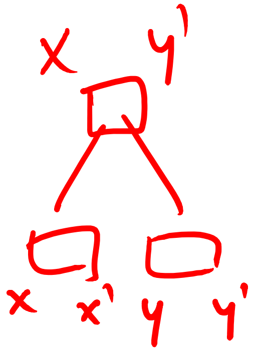
Suppose we have undeclared variable ‘i’ in a program of 100K lines

What error message would you like compiler to give?

- An ocurrence of variable ‘i’ not declared
- An ocurrence of variable ‘i’ in procedure P not declared
- Variable ‘i’ undeclared at line 514, position 12 🡐

How to emit this error message if we only have a syntax trees?

- Abstract syntax tree nodes store positions in file where they begin!
- For identifier nodes: allows reporting variable uses
  - Variable 'i' in line 11, column 5 undeclared
- For other nodes, supports useful for type errors, e.g. could report for (x + y) * (!ok)
  - Type error in line 13,
  - expression in line 13, column 11-15, has type bool, but int is expected



Constructing trees with positions:

- obtain position from lexer when parsing beginning of tree node
- save this position in the constructed tree
- can also save end positions

What is important is to save information for leaves

- information for other nodes can be approximated using information in leaves

Example: find program result, symbols, scopes

```
class Example {
  boolean x;
  int y;
  int z;
  int compute(int x, int y) {
        int z = 3;
        return x + y + z;
  }
  public void main() {
        int res;
        x = true;
        y = 10;
        z = 17;
        res = compute(z, z+1);
        System.out.println(res);
  }
}
```

Scope of a variable = part of the program where it is visible

Draw an arrow from occurrence of each identifier to the point of its declaration.

For each declaration of identifier, identify where the identifier can be referred to (its scope).

Name analysis:

- computes those arrows
  = maps, partial functions (math)
  = environments (PL theory)
  = symbol table (implementation)
- report some simple semantic errors

We usually introduce **symbols** for things denoted by identifiers.
Symbol tables map identifiers to symbols.

Name Analysis: Problems it Detects

- a class is defined more than once: **class A { ...} class B { ... } class A { ... }**
- a variable is defined more than once: **int x; int y; int x;**
- a class member is overriden without **override** keyword:
  class A { int x; ... } class B extends A { int x; ... }
- a method is **overloaded** (forbidden in Tool):
  class A { def f(B x) {} def f(C x) {} ... }
- a method argument is shadowed by a local variable declaration (forbidden in Java, Tool):
  def (x:Int) { var x : Int; ...}
- two method arguments have the same name: **def (x:Int,y:Int,x:Int) { ... }**
- a class name is used as a symbol (as parent class or type, for instance) but is not declared:
  class A extends Objekt {}
- an identifier is used as a variable but is not declared:
  def(amount:Int) { total = total + ammount }

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \arrow[ll, bend left] \end{tikzcd}$$

- the inheritance graph has a cycle: **class A extends B {} class B extends C {} class C extends A**

To make it efficient and clean to check for such errors, we associate mapping from each identifier to the symbol that the identifier represents.

- We use Map data structures to maintain this mapping
- The rules that specify how declarations are used to construct such maps are given by ***scoping* rules of the programming language.**

Usual **static** scoping: What is the result?

```
class World {
 int sum;
 int value;
 void add() {
    sum = sum + value;
    value = 0;
 }
 void main() {
     sum = 0;
     value = 10;
     add();
     if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);  1
        print("sum = ", sum);  10
     }
     print("outer value = ", value);   0
  }
}
```

Identifier refers to the symbol that was declared closest to the place **in program text** (thus "static").

We will assume static scoping unless otherwise specified.

Renaming Statically Scoped Program

```
class World {
 int sum;
 int value;
 void add(int foo) {
    sum = sum + value;
    value = 0;
 }
 void main() {
     sum = 0;
     value = 10;
     add();
     if (sum % 3 == 1) {
       int value1;
       value1 = 1;
       add(); // cannot change value1
       print("inner value = ", value1); 1
       print("sum = ", sum);  10
     }
     print("outer value = ", value);  0
  }
}
```

Identifier refers to the symbol that was declared closest to the place **in program text** (thus "static").

We will assume static scoping unless otherwise specified.

Property of static scoping:
Given the entire program, we can **rename variables** to avoid any shadowing (**make** all vars **unique**!)

Dynamic scoping: What is the result?

```
class World {
 int sum;
 int value;
 void add() {
    sum = sum + value;
    value = 0;
 }
 void main() {
     sum = 0;
     value = 10;
     add();
     if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);  0
        print("sum = ", sum);  11
     }
     print("outer value = ", value);  0
  }
}
```

Symbol refers to the variable that was most **recently declared within program execution.**

Views variable declarations as executable statements that establish which symbol is considered to be the ‘current one’. (Used in old LISP interpreters.)

Translation to normal code: access through a dynamic environment.

Dynamic scoping translated using global map, working like stack

```
class World {
 int sum;
 int value;
 void add() {
    sum = sum + value;
    value = 0;
 }
 void main() {
     sum = 0;
     value = 10;
     add();
     if (sum % 3 == 1) {
        int value;
        value = 1;
        add();
        print("inner value = ", value);  0
        print("sum = ", sum);   11
     }
     print("outer value = ", value);  0
  }
}
```

```
class World {
 pushNewDeclaration('sum);
 pushNewDeclaration('value);
 void add(int foo) {
    update('sum, lookup('sum) + lookup('value));
    update('value, 0);
 }
 void main() {
     update('sum, 0);
     update('value,10);
     add();
     if (lookup('sum) % 3 == 1) {
        pushNewDeclaration('value);
        update('value, 1);
        add();
        print("inner value = ", lookup('value));
        print("sum = ", lookup('sum));
        popDeclaration('value)
     }
     print("outer value = ", lookup('value));
  }
}
```

Object-oriented programming has scope for each object, so we have a nice controlled alternative to dynamic scoping (objects give names to scopes).

How the **symbol map** changes in case of **static** scoping

```
class World {
 int sum;  int value;
 // value → int, sum → int
 void add(int foo) {
     // foo → int, value → int, sum → int
     string z;
     // z → string, foo → int, value → int, sum → int
     sum = sum + value; value = 0;
 }
 // value → int, sum → int
 void main(string bar) {
     // bar → string, value → int, sum → int
     int y;
     // y → int, bar → string, value → int, sum → int
     sum = 0;
     value = 10;
     add();
     // y → int, bar → string, value → int, sum → int
     if (sum % 3 == 1) {
       string value;
       // value → string, y → int, bar → string, sum → int
       value = 1;
       add();
       print("inner value = ", value);
       print("sum = ", sum); }
     // y → int, bar → string, value → int, sum → int
     print("outer value = ", value);
} }
```

Outer declaration **int value** is shadowed by inner declaration **string value**

Map becomes bigger as we enter more scopes, later becomes smaller again

Imperatively: need to make maps bigger, later smaller again.

Functionally: immutable maps, keep old versions.

Formalism for Name Analysis

NOTATION FOR MAPS

Mathematical notion of map $f : A \rightharpoonup B$ is a partial function, that is, a function from a subset of A to B.

- $f \subseteq A \times B$
- $\forall x.\forall y_1.\forall y_2.\ (x, y_1) \in f \wedge (x, y_2) \in f \rightarrow y_1 = y_2$

We define $dom(f) = \{x \mid \exists y.(x, y) \in f\}$

Key operation is function update

$f[k := v] = \{(x, y) \mid (x = k \wedge y = v) \vee (x \neq k \wedge (x, y) \in f)\}$

If the value was defined before, now we redefine it.

A generalization of update is overriding one map by another:

$f \oplus g = \{(x, y) \mid (x, y) \in g \vee (x \notin dom(g) \wedge (x, y) \in f\}$

Sometimes we denote map $\{(k_1, v_1), \ldots, (k_n, v_n)\}$ by $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$

Is $f \oplus g = g \oplus f$?

- $\{x \mapsto b, z \mapsto a\} \oplus \{x \mapsto c\} = \{x \mapsto c, z \mapsto a\}$
- $\{x \mapsto c\} \oplus \{x \mapsto b, z \mapsto a\} = \{x \mapsto b, z \mapsto a\}$

Checking that each variable is declared

$\Gamma = \{ (x_1, T_1), \ldots (x_n, T_n) \}$ - environment (symbol table)

identifier ↗ x_1; symbol (type, ...) ↖ T_1

$$\boxed{\Gamma \vdash e}$$

"e uses only variables declared in Γ"

$\Gamma = \{ (x, int), (y, string), (z, int) \}$

then:

$\Gamma \vdash x + (z + 1)$

$\Gamma \vdash x = x + 1$

Rules for Checking Variable Use

$$\boxed{\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 + e_2}}$$

$$\boxed{\frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 * e_2}}$$

$$\boxed{\frac{(x,\_) \in \Gamma \quad \Gamma \vdash e}{\Gamma \vdash x = e}}$$

$$\boxed{\frac{(x,\_) \in \Gamma}{\Gamma \vdash x} \quad \text{use of a variable}}$$

$$\boxed{\frac{\Gamma \vdash s \quad \Gamma \vdash \bar{s}}{\Gamma \vdash s ; \bar{s}}}$$

s - statement

$\bar{s}$ - statement sequence

Local Block Declarations Change Γ

$$\frac{\Gamma[x := int] \vdash \bar{s}}{\Gamma \vdash (int\ x); \bar{s}}$$

$\Gamma = \{(z, int)\}$

$\Gamma[x := int] = \{(z, int), (x, int)\}$

$$\frac{\Gamma[x := int] \vdash x = z + 2}{\Gamma \vdash int\ x;\ x = z + 2}$$

Method Parameters are Similar

$$\frac{\Gamma \oplus \{(x_1, T_1), \ldots, (x_n, T_n)\} \vdash \bar{s}}{\Gamma \vdash T\; m\; (T_1\, x_1, \ldots, T_n\, x_n)\; \{\, \bar{s}\, \}}$$

$\Gamma = \{(sum, int), (value, int)\}$

$$\frac{\Gamma \oplus \{(foo, int)\} \vdash sum = sum + foo;}{\Gamma \vdash \text{void add (int foo) \{ sum = sum + foo; \}}}$$

```
class World {
  int sum;
  int value;
  void add(int foo) {
     sum = sum + foo;
  }
}
```

Symbol Table (Γ) Contents

- Map identifiers to the symbol with relevant information about the identifier
- All information is derived from syntax tree - symbol table is for efficiency
  - in old one-pass compilers there was only symbol table, no syntax tree
  - in modern compiler: we could always go through entire tree, but symbol table can give faster and easier access to the part of syntax tree, or some additional information
- Goal: efficiently supporting phases of compiler
- In the name analysis phase:
  - finding which identifier refers to which definition
  - we store *definitions*
- What kinds of things can we define? What do we need to know for each ID?

variables (globals, fields, parameters, locals):

  - need to know types, positions - for error messages
  - later: memory layout. To compile x.f = y into memcopy(addr_y, addr_x+6, 4)
    - e.g. 3rd field in an object should be stored at offset e.g. +6 from the address of the object
    - the size of data stored in x.f is 4 bytes
  - sometimes more information explicit: whether variable local or global

methods, functions, classes: recursively have with their own symbol tables

Different Points, Different Γ

$\Gamma_0 = \{(sum, int), (count, int)\}$

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\leftarrow \Gamma_1 = \Gamma_0[foo := int]$

$\leftarrow \Gamma_0$

$\leftarrow \Gamma_1 = \Gamma_0[bar := int]$

Imperative Way: Push and Pop

```
class World {
  int sum;
  void add(int foo) {
     sum = sum + foo;
  }
  void sub(int bar) {
     sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

change table, record change

Γ_0

revert changes from table

$\Gamma_1 = \Gamma_0 [bar := int]$

change table, record change

revert changes from table

Imperative Symbol Table

- Hash table, mutable Map[ID,Symbol]
- Example:
  - hash function into array
  - array has linked list storing (ID,Symbol) pairs
- Undo stack: to enable entering and leaving scope
- Entering new scope (function,block):
  - add beginning-of-scope marker to undo stack
- Adding nested declaration (ID,sym)
  - lookup old value symOld, push old value to undo stack
  - insert (ID,sym) into table
- Leaving the scope
  - go through undo stack until the marker, restore old values

Functional: Keep Old Version

```
class World {
  int sum;
  void add(int foo) {
    sum = sum + foo;
  }
  void sub(int bar) {
    sum = sum - bar;
  }
  int count;
}
```

$\Gamma_0 = \{(sum, int), (count, int)\}$

$\Gamma_1 = \Gamma_0 [foo := int]$

create new Γ_1, keep old Γ_0

Γ_0

$\Gamma_2 = \Gamma_0 [bar := int]$

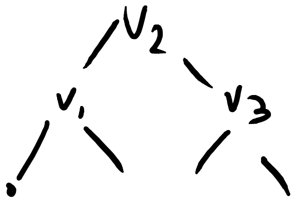
create new Γ_2, keep old Γ_0

Functional Symbol Table Implemented

- Typical: Immutable Balanced Search Trees

```
sealed abstract class BST
case class Empty() extends BST
case class Node(left: BST, value: Int, right: BST) extends BST
```

Simplified. In practice, BST[A], store Int key and value A

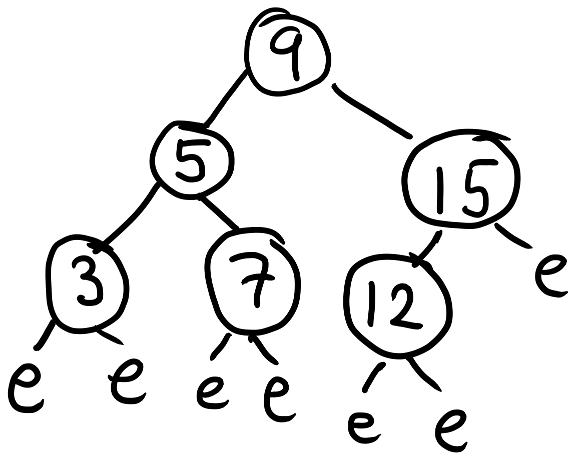


- Updating returns new map, keeping old one
  - lookup and update both *log(n)*
  - update creates new path (copy *log(n)* nodes, share rest!)
  - memory usage acceptable

Lookup

```
def contains(key: Int, t : BST): Boolean = t match {
    case Empty() => false
    case Node(left,v,right) => {
        if (key == v) true
        else if (key < v) contains(key, left)
        else contains(key, right)
    }
}
```

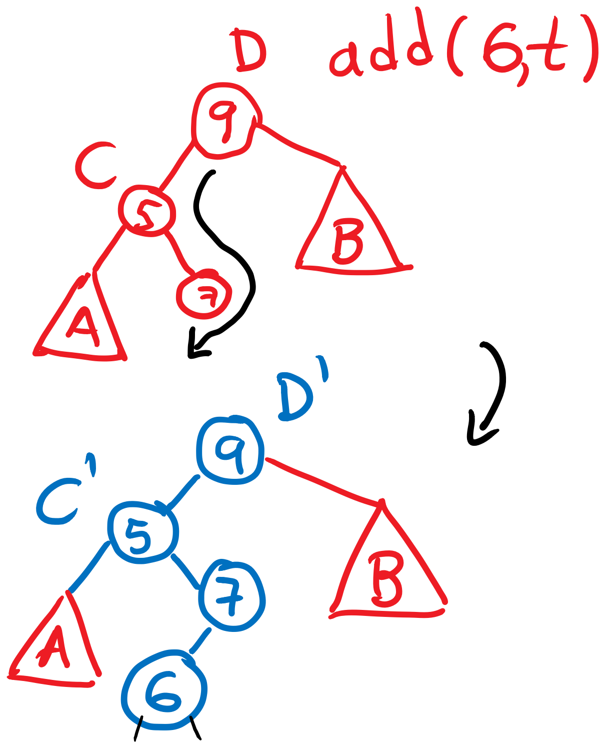
Running time bounded by tree height.



contains(6,t) ?

Insertion

```
def add(x : Int, t : BST) : Node = t match {
  case Empty() => Node(Empty(),x,Empty())
  case t @ Node(left,v,right) => {
   if (x < v) Node(add(x, left), v, right)
   else if (x==v) t
   else Node(left, v, add(x, right))
  }
}
```

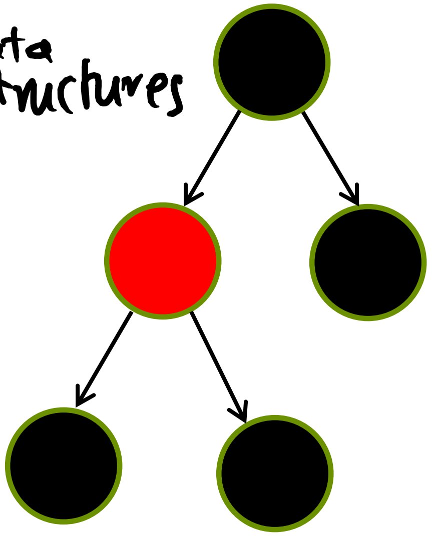


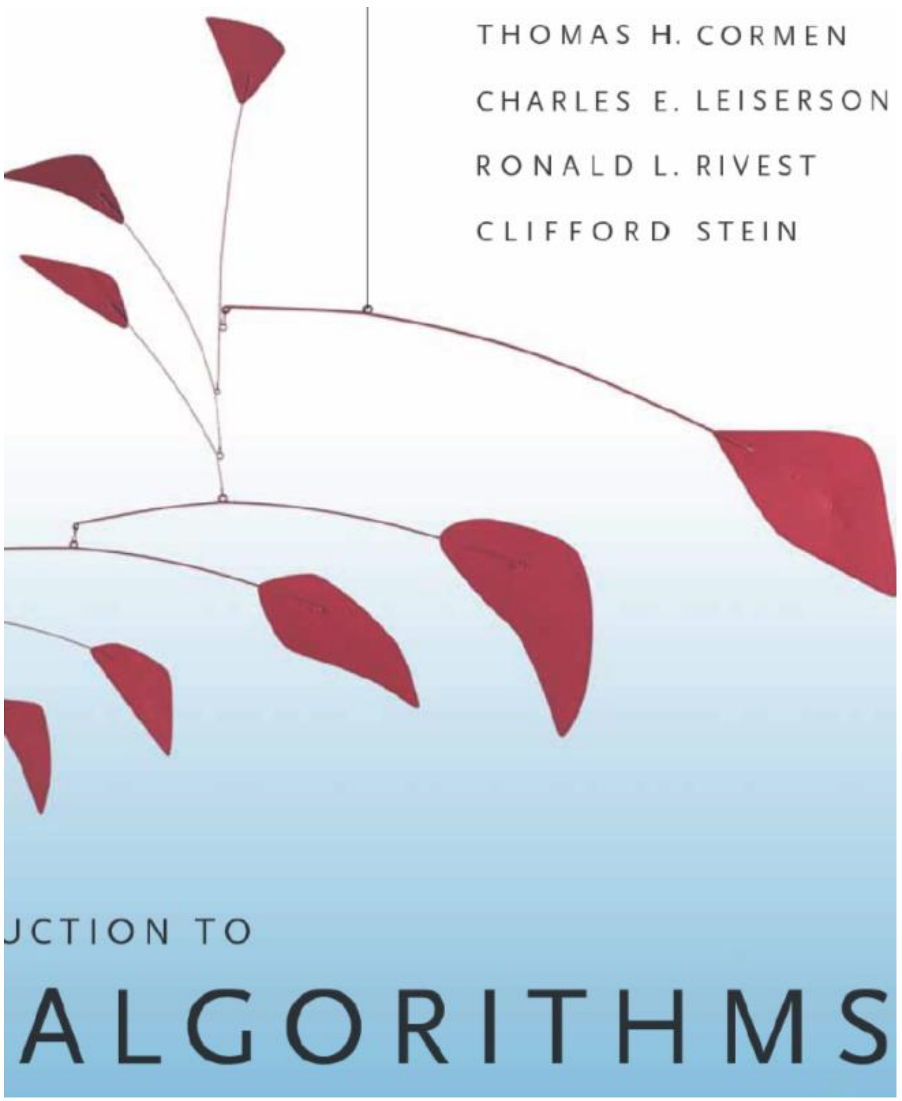
Both add(x,t) and t remain accessible.

Running time and newly allocated nodes bounded by tree height.

Chris Okasaki : Purely Functional Data Structures

Balanced Trees: Red-Black Trees





THOMAS H. CORMEN

CHARLES E. LEISERSON

RONALD L. RIVEST

CLIFFORD STEIN

UCTION TO

ALGORITHMS

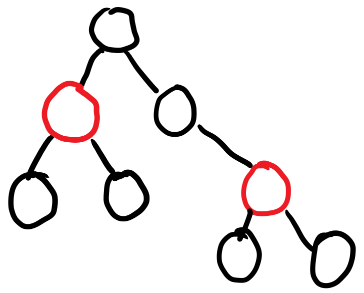
Balanced Tree: Red Black Tree

Goals:

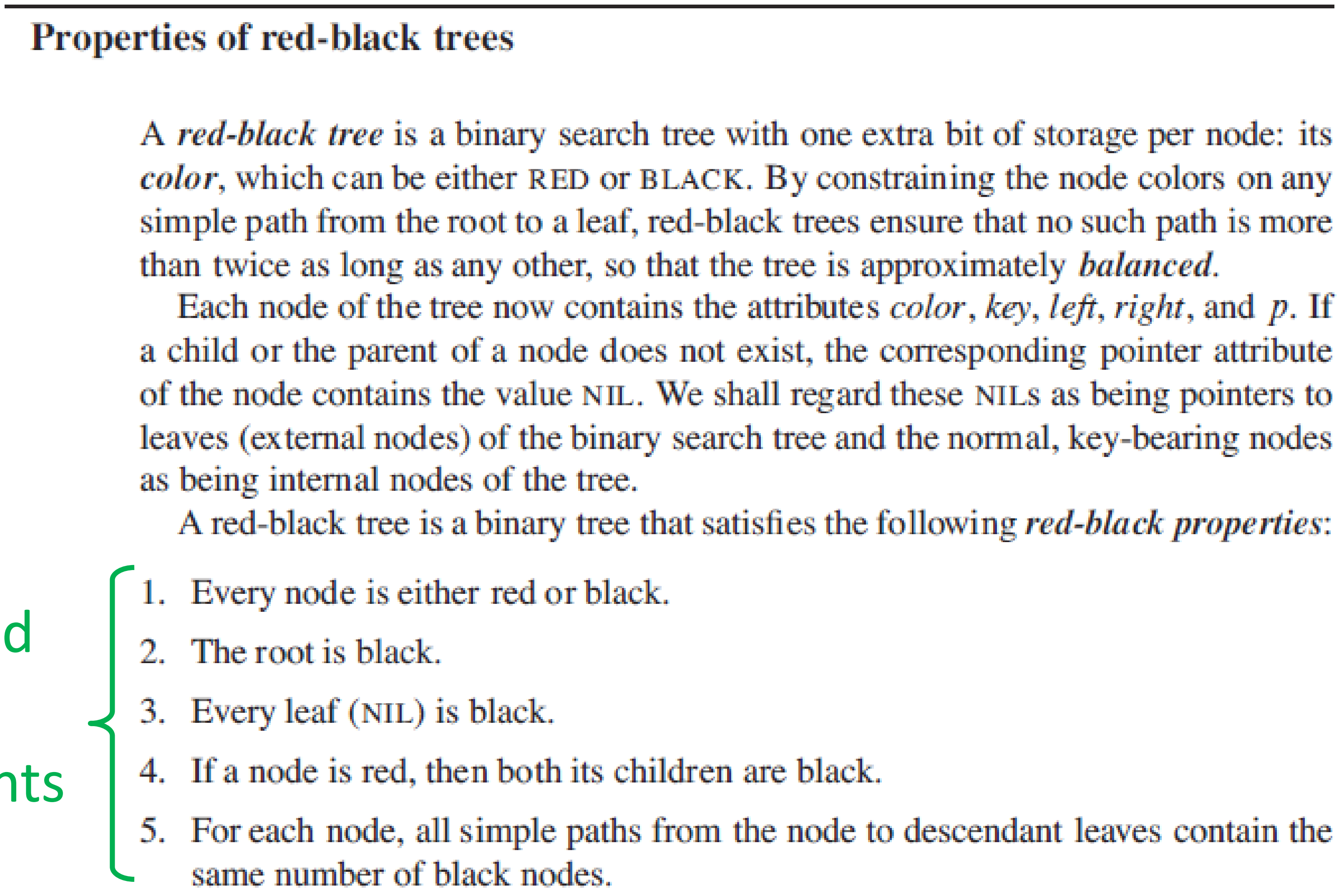
- ensure that tree height remains at most log(size)
  add(1,add(2,add(3,...add(n,Empty())...))) ~ linked list ✗
- preserve efficiency of individual operations:
  rebalancing arbitrary tree: could cost O(n) work

Solution: maintain mostly balanced trees: height still O(log size)

```
sealed abstract class Color
case class Red() extends Color
case class Black() extends Color
```



```
sealed abstract class Tree
case class Empty() extends Tree
case class Node(c: Color,left: Tree,value: Int, right: Tree)
                    extends Tree
```



Properties of red-black trees

A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately ***balanced***.

Each node of the tree now contains the attributes *color*, *key*, *left*, *right*, and *p*. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following ***red-black properties***:

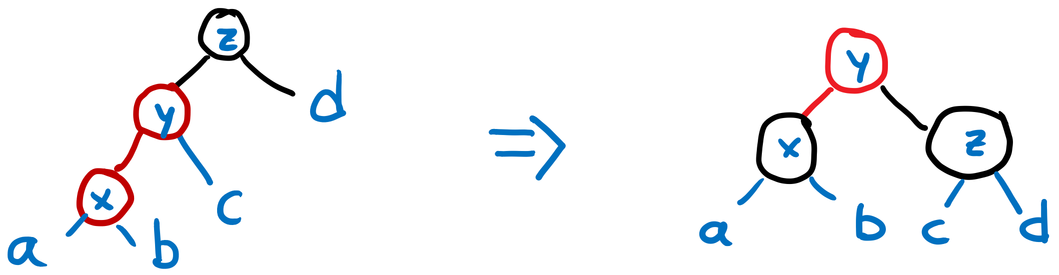
balanced tree constraints

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

From 4. and 5.: tree height is O(log size).
Analysis is similar for mutable and immutable trees.
for immutable trees: see book by Chris Okasaki

Balancing

```
def balance(c: Color, a: Tree, x: Int, b: Tree): Tree = (c,a,x,b) match {
  case (Black(),Node(Red(),Node(Red(),a,xV,b),yV,c),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
```



```
  case (Black(),Node(Red(),a,xV,Node(Red(),b,yV,c)),zV,d) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),Node(Red(),b,yV,c),zV,d)) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (Black(),a,xV,Node(Red(),b,yV,Node(Red(),c,zV,d))) =>
  Node(Red(),Node(Black(),a,xV,b),yV,Node(Black(),c,zV,d))
  case (c,a,xV,b) => Node(c,a,xV,b)
}
```

Insertion

```
def add(x: Int, t: Tree): Tree = {
  def ins(t: Tree): Tree = t match {
    case Empty() => Node(Red(),Empty(),x,Empty())
    case Node(c,a,y,b) =>
      if (x < y) balance(c, ins(a), y, b)
      else if (x == y) Node(c,a,y,b)
      else balance(c,a,y,ins(b))
  }
  makeBlack(ins(t))
}
def makeBlack(n: Tree): Tree = n match {
    case Node(Red(),l,v,r) => Node(Black(),l,v,r)
    case _ => n
}
```

Modern object-oriented languages (e.g. Scala) support abstraction and functional data structures. Just use Map from Scala.

Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```
object MyClass {
  val x = 5
  def foo(z: Int): Int = { x + z }
  def bar(y: Int): Int = {
     val x = 1; val z = 2
     foo(y)
  }
  def main() {
    val x = 7
    println(foo(bar(3)))
  }
}
```