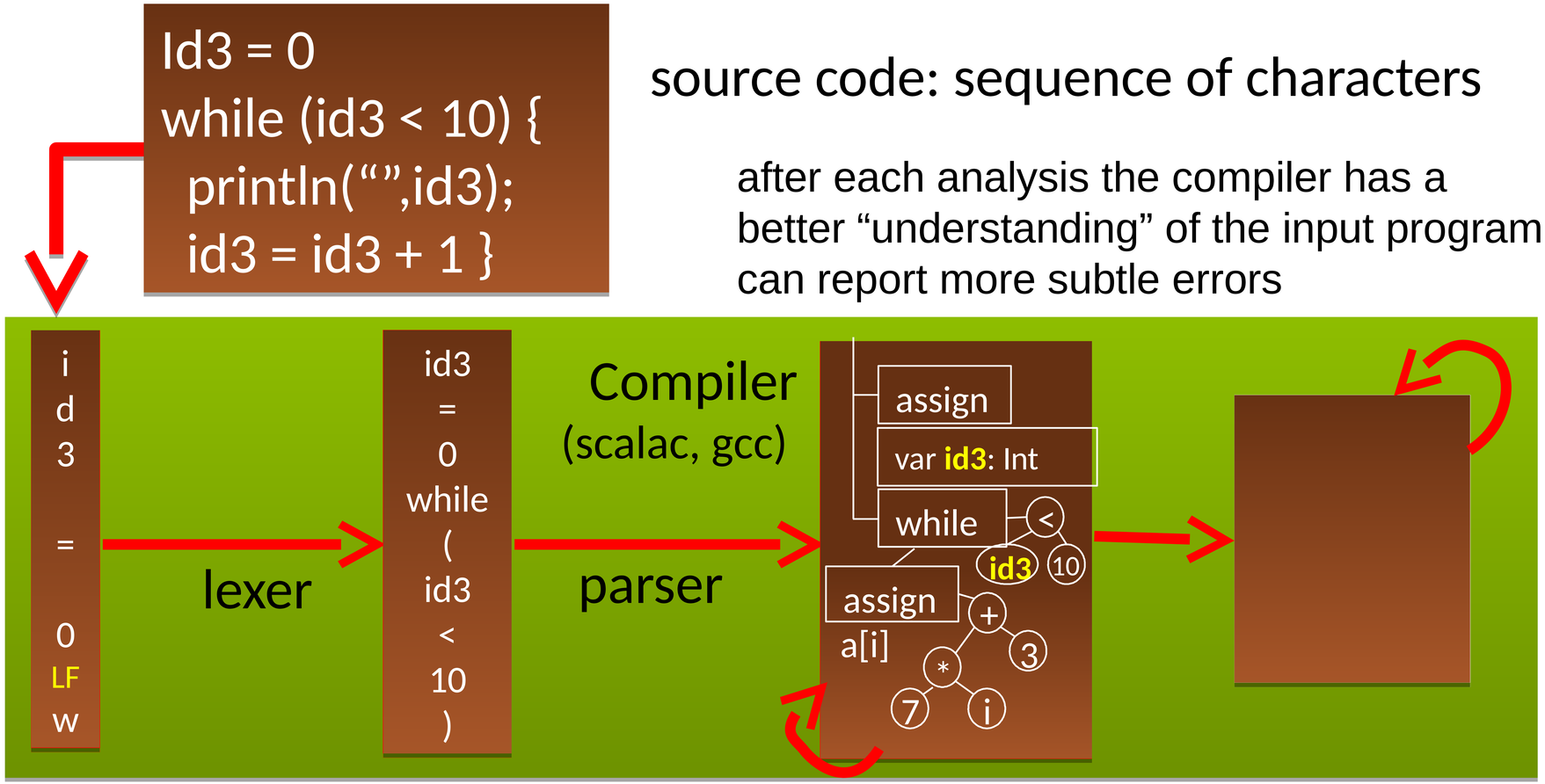

Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code: sequence of characters
after each analysis the compiler has a better “understanding” of the input program; can report more subtle errors
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
Compiler
(scalac, gcc)
parser
assign
var id3: Int
while
<
id3
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees
Name Analysis:
making sense of trees;
converting them into graphs:
connect identifier uses and declarations

Reporting Errors

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 identifier
- Type analyzer:
  applying function to arguments of wrong type
- Data-flow analyzer:
  variable read before written, division by zero

Name Analysis Problems Reported: 1

- 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) { … }

Name Analysis Problems Reported: 2

- 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 }

- 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.**

Storing and Using Tree Positions

Showing Good Errors with Syntax Trees

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

Which error message would you prefer to see from the compiler?

- An ocurrence of variable ‘i’ not declared (which variable? where?)
- An ocurrence of variable ‘i’ in procedure P not declared
- Variable ‘i’ undeclared at line 514, position 12 (and IDE points you there)🗲

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

- Abstract syntax tree nodes store positions within file
- 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**, expected **Int** instead

Showing Good Errors with Syntax Trees

Constructing trees with positions:

- Lexer records positions for tokens
- Each subtree in AST corresponds to some parse tree, so it has first and last token
- Get positions from those tokens
- Save these positions in the constructed tree

What is important is to save information for leaves

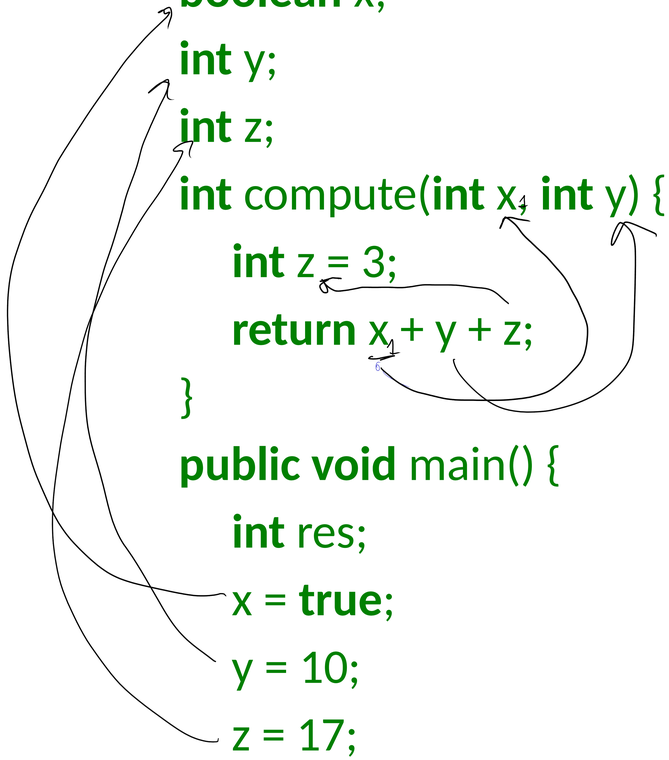
- information for other nodes can often be approximated using information in the leaves

Continuing Name Analysis:

Scope of Identifiers

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.

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 structure** (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 structure** (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).

Good Practice for Scoping

- Static scoping is almost universally accepted in modern programming language design
- It is the approach that is usually easier to reason about and easier to **compile**, since we do not have names at compile time and compile each code piece separately
- Still, various ad-hoc language designs emerge and become successful
  - LISP implementations took dynamic scoping since it was simpler to implement for higher-order functions
  - Javascript