CS558
Programming Languages

Fall 2023
Lecture 2a

Andrew Tolmach
Portland State University

This week’s lab: Expressions

- Inspired by familiar mathematical notation
- Usually have recursive (tree-like) structure
- Can be used to define values in many domains
  - numbers, booleans, strings, lists, sets, etc.
- “Declarative” syntax: tells what to compute rather than how
- Abstracts away from evaluation order* and use of temporaries
  - compare with, e.g., stack machine

* to some extent: depends on language

Imperative Languages

- Most commonly-used languages are imperative

- Consist of sequence of commands that alter the state of the world

- State = values of program variables and external environment (e.g. files, screen, etc.)

http://smarteregg.com/dont-tell-me-what-to-do-now-show-me-what-i-need-to-be-doing/

Running Imperative Programs

- High-level imperative languages mimic style of the underlying Von Neumann machine architecture
  - machine programs are sequences of instructions that modify registers and memory locations
- Compiling imperative languages to machine code is relatively straightforward
  - variables are mapped to machine locations
  - commands (operations) are mapped to (multiple) machine instructions

Reactive Programs

- Imperative languages are also natural for writing reactive programs that interact with the real world
- Examples:
  - Reading mouse clicks and modifying the contents of a display
  - Communicating data on a network link
  - Controlling a set of sensors and relays in an external device
- Often structured as event-response loops

Statements are Commands

- Elementary (atomic) statements
  - Assignment
  - I/O operations
  - Function/Procedure calls
    - Atomic from perspective of caller
- Compound statements
  - Built recursively from sub-statements, forming tree-like structure

Assignment

- Most primitive command: store a value into a location

```
a := 42
```

- In simplest form, location is associated with a variable
  - but might be an array or record element, etc.

```
a[x+2] := 42
```

- In most languages, a variable name means different things on the left-hand side (LHS) and right-hand side (RHS) of an assignment.

```
a := a + 5
```

  - On LHS, name denotes the location of the variable, into which the value of the RHS expression is to be stored. Here we say name is an l-value.
  - On RHS, name denotes the current value contained in the location, i.e. it indicates an implicit dereference operation. Here we say the name is an r-value.

Assignment Expressions

`b := (a := 42)`

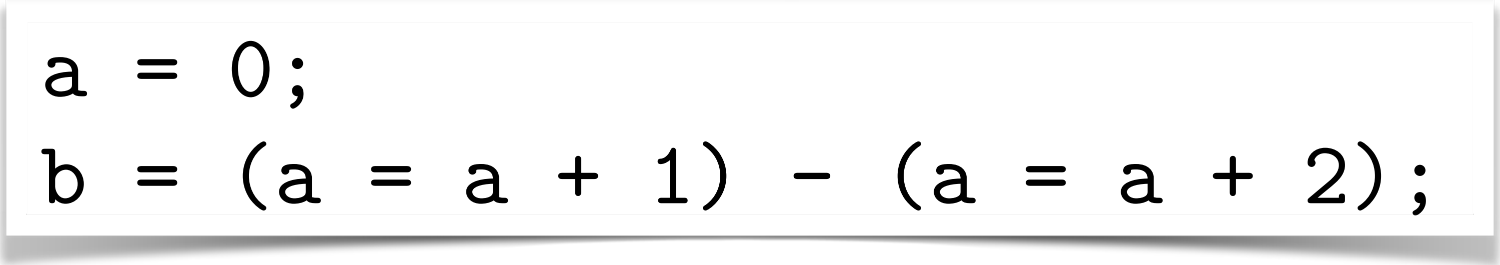
`f(c := 10)`

- In some languages, assignment is an expression
  - and expressions can act as atomic statements
  - But every expression must define a value! Common choices for the value of an assignment:
    - value of LHS after assignment
    - special “no information” value e.g., in Scala: `():Unit`
  - C/C++/Java popularized use of plain = for assignment and == for relational equality: a truly bad idea, because both form expressions and they are easy to confuse

Order of Operations

- We’ve noted that order of operations for expressions is usually under-specified
  - Parse tree doesn’t completely fix order
- But this causes problems if expressions can be assignments:

ANSI C99



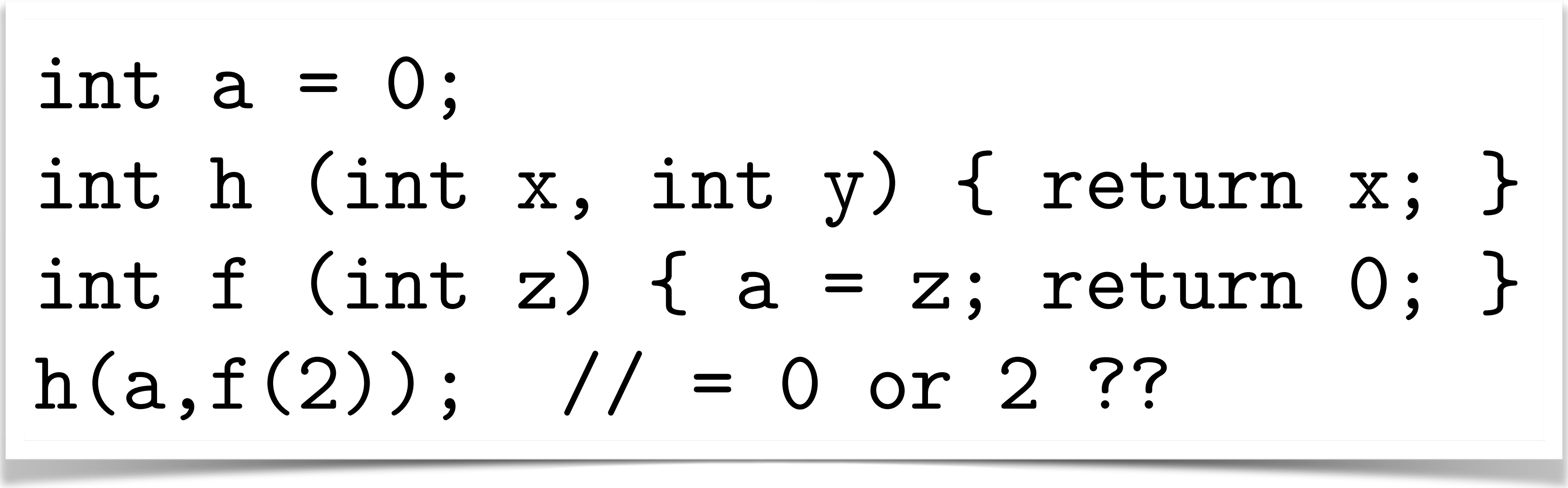
```
a = 0;
b = (a = a + 1) - (a = a + 2);
```

- What is the result in b?
- It can be anything! This C program has “undefined behavior” and the compiler can generate anything it wants (for the entire program!)
  - …or the compiler could give a warning or error message, but many compilers do not.

Hidden side-effects

- Even without explicit assignment expressions, expression evaluation order can affect behavior:

ANSI C99

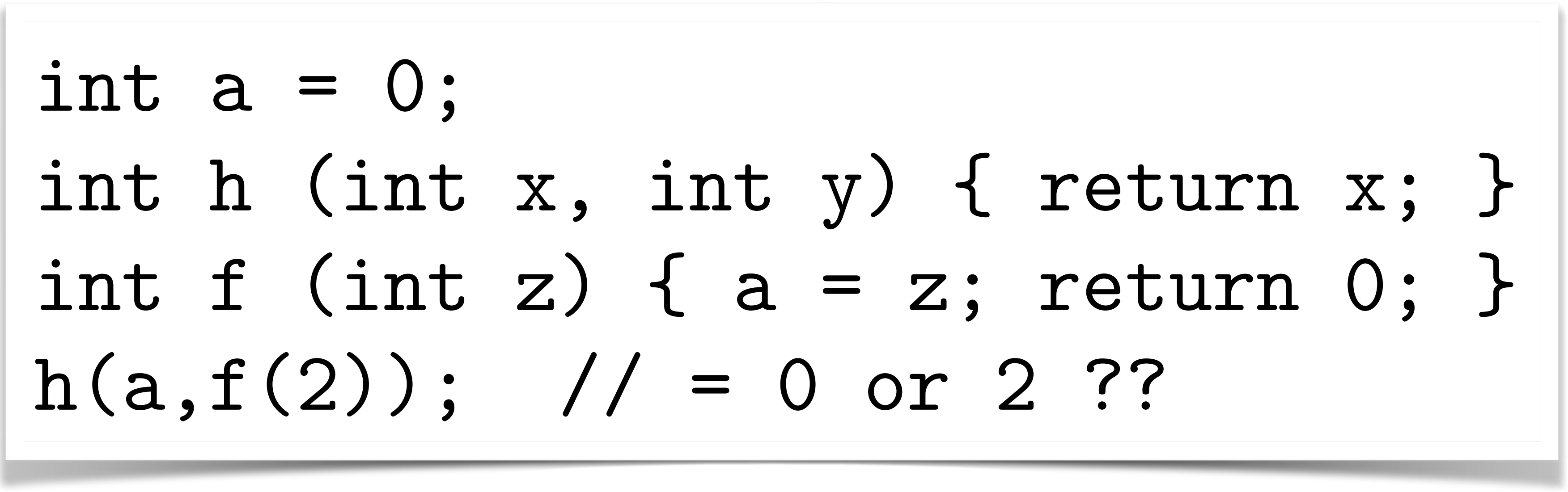


```
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2));  // = 0 or 2 ??
```

- Answer depends on evaluation order for function parameters, which is compiler-dependent (though “unspecified” rather than “undefined”)
- This flexibility may let compiler generate more efficient code
- But most modern languages are moving towards precise specification of order (e.g. left-to-right)

Imperative code is infectious

ANSI C99



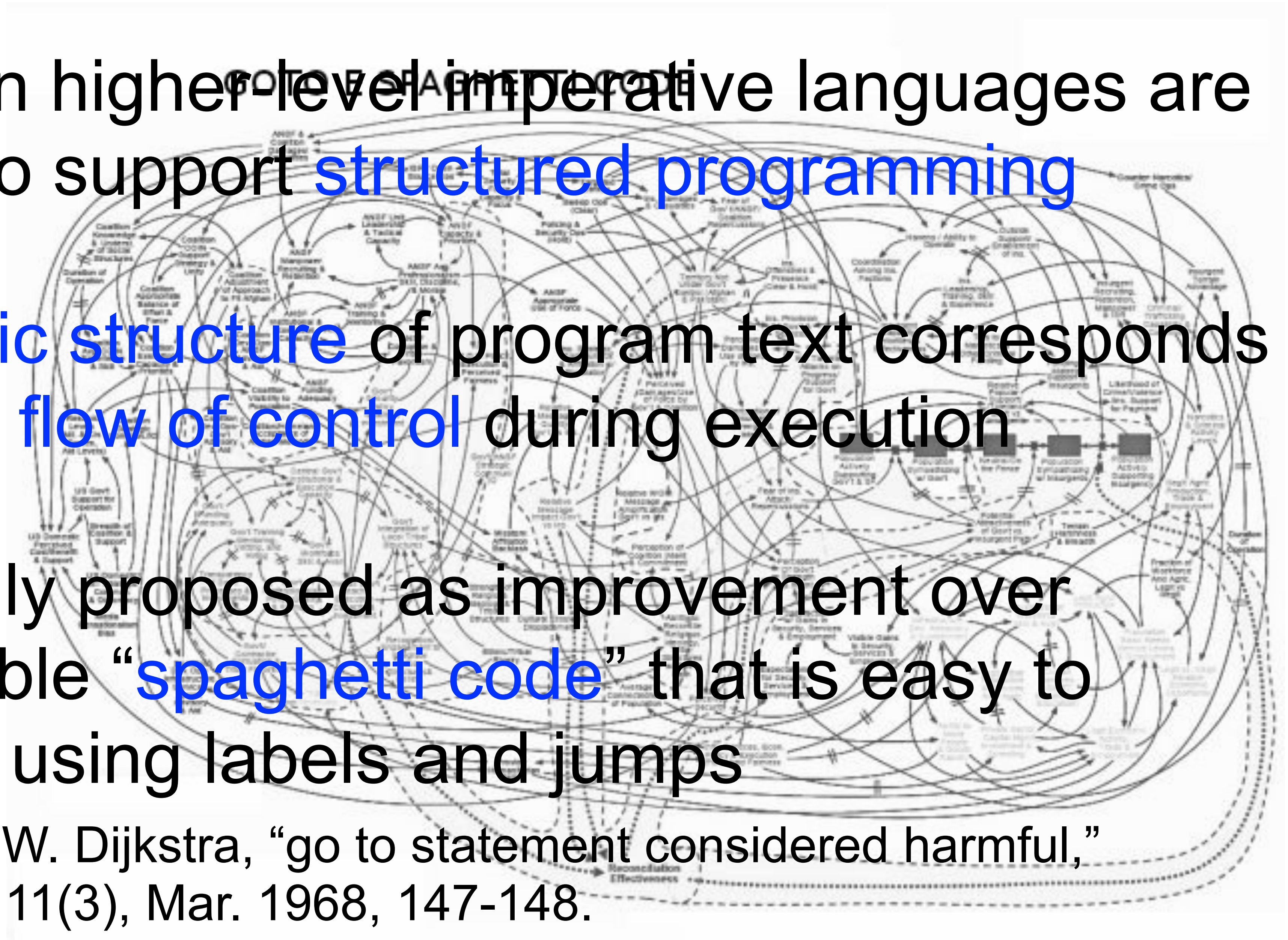
```
int a = 0;
int h (int x, int y) { return x; }
int f (int z) { a = z; return 0; }
h(a,f(2));  // = 0 or 2 ??
```

- Root of problem is that imperative code can be hidden within function definitions (“side-effects”)

- If any part of the code might be imperative, we must worry about order of evaluation in all parts of the code

- In a few languages, the type system helps us distinguish functions that have side-effects from "pure" ones that don't

Structured Control Flow



- All modern higher-level imperative languages are designed to support structured programming
  - Syntactic structure of program text corresponds to dynamic flow of control during execution
  - Originally proposed as improvement over unreadable “spaghetti code” that is easy to produce using labels and jumps

    Edsger W. Dijkstra, “go to statement considered harmful,” CACM, 11(3), Mar. 1968, 147-148.

blogbv2.altervista.org

Small set of statement kinds

- Use small collection of (recursively defined) compound statements to describe control flow

- Sequential composition: do a sequence of commands

```
(Java) { x = 2; y = x + 4;}
(Pascal) begin x := 2; y := x + 4; end
```

- Selection: do one of several alternative commands

```
(Java) if (x < 0) y = x + 1; else z = y + 2;
```

- Iteration: do a command repeatedly

```
(Java) while (x > 10) output(x--);
(Pascal) for x := 1 to 12 do output(x*2);
```

Sequential composition

- Simplest way to combine commands: just write one after another
- Order obviously matters!
- (What about parallel composition?)
- Can also have sequential composition of expressions
  - $e_1 ; e_2$ means: evaluate e_1; throw away the result; then evaluate e_2
  - obviously only interesting if e_1 has side-effects

Selection: `if`

- Basic selection statement based on booleans

$$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$

compiles to

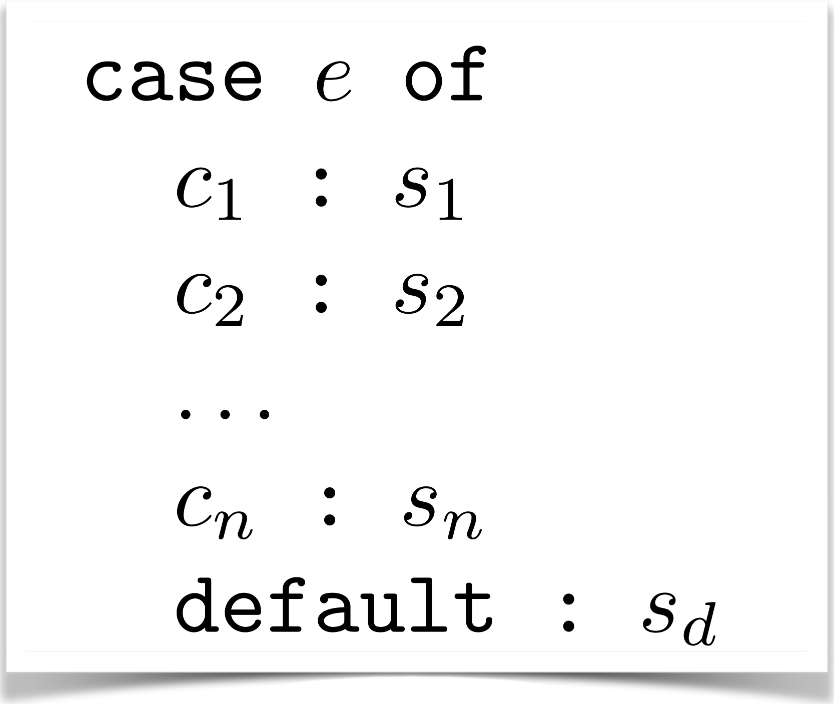
Structured statements have simple equivalents in terms of labels + jumps

```
         evaluate $e$ into $t$
         cmp $t$,true
         brneq $l_1$
         $s_1$
         br $l_2$
$l_1$:    $s_2$
$l_2$:
```

pseudo assembly code

Selection: `case`

- Generalizes boolean conditionals to types with larger domains

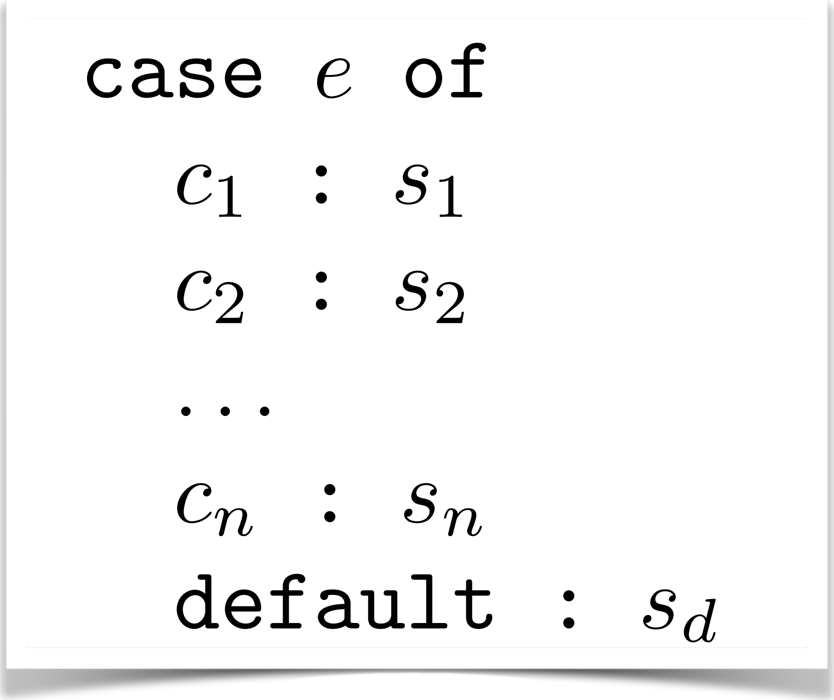


```
case $e$ of
   $c_1$ : $s_1$
   $c_2$ : $s_2$
   ...
   $c_n$ : $s_n$
   default : $s_d$
```

- Note that the *c*c_i are constants

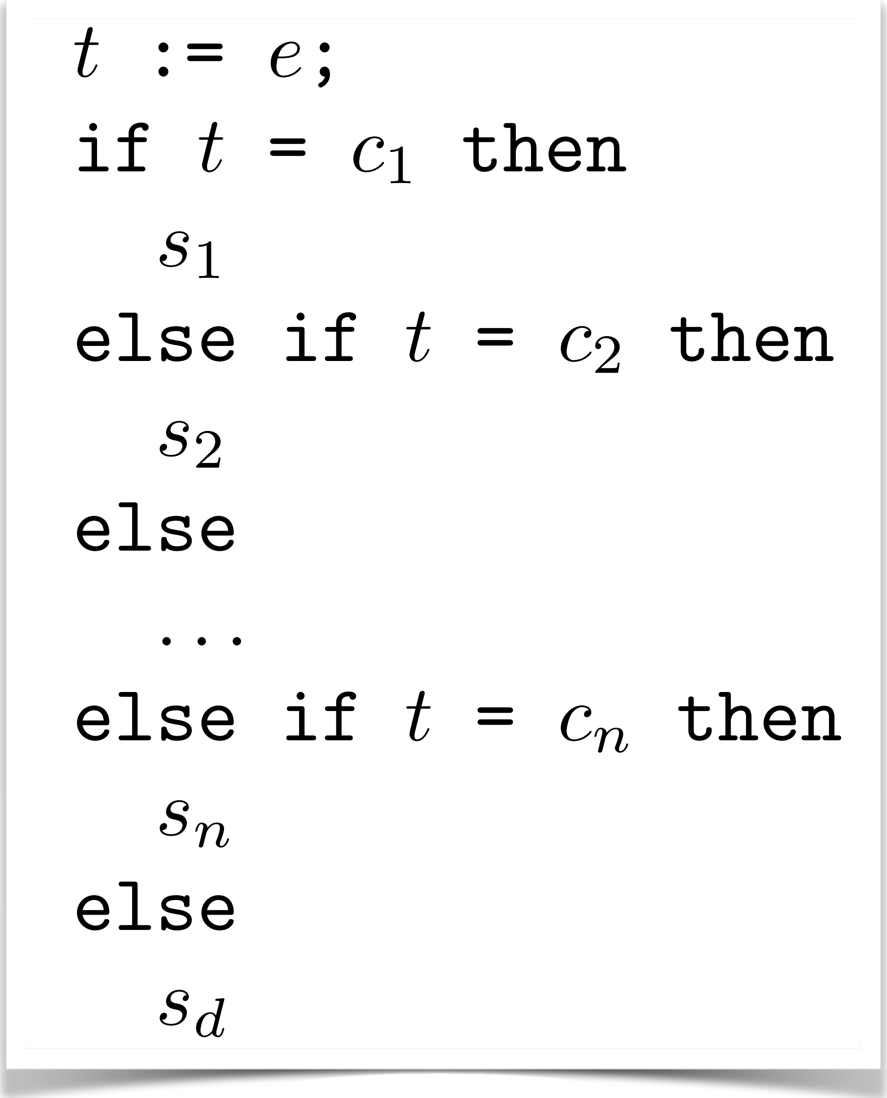
- Choice of most efficient compilation method depends on density of the c_i within the domain of possible values for *e* and on whether *e*'s type is ordered

Sparse `case` compilation



```
case $e$ of
   $c_1$ : $s_1$
   $c_2$ : $s_2$
   ...
   $c_n$ : $s_n$
   default : $s_d$
```

is equivalent to

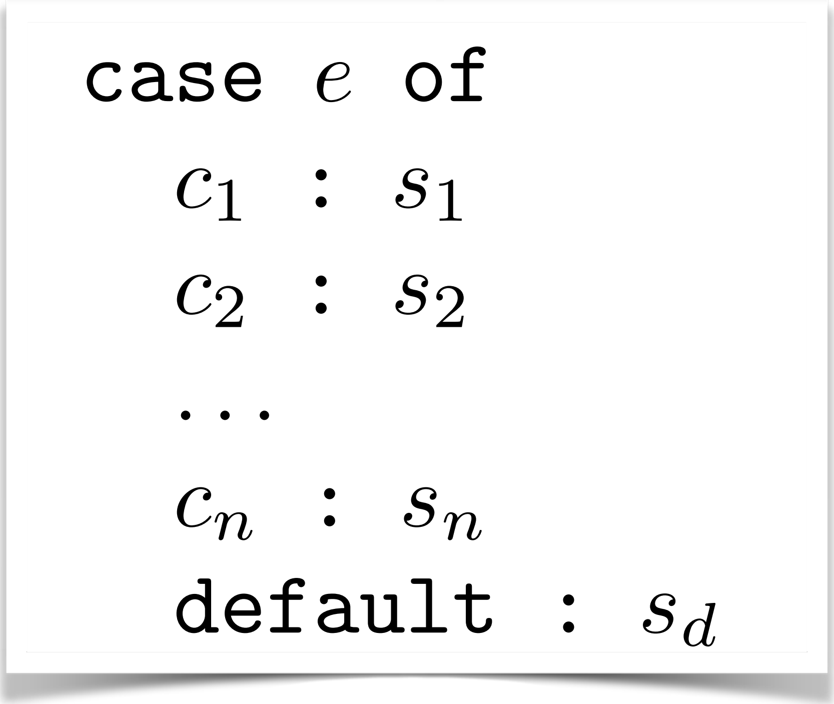


```
$t$ := $e$;
if $t$ = $c_1$ then
   $s_1$
else if $t$ = $c_2$ then
   $s_2$
else
   ...
else if $t$ = $c_n$ then
   $s_n$
else
   $s_d$
```

- This is just a linear search (O(n) time)
- If *e*’s type is ordered, we can do better with a binary search (O(log n) time)

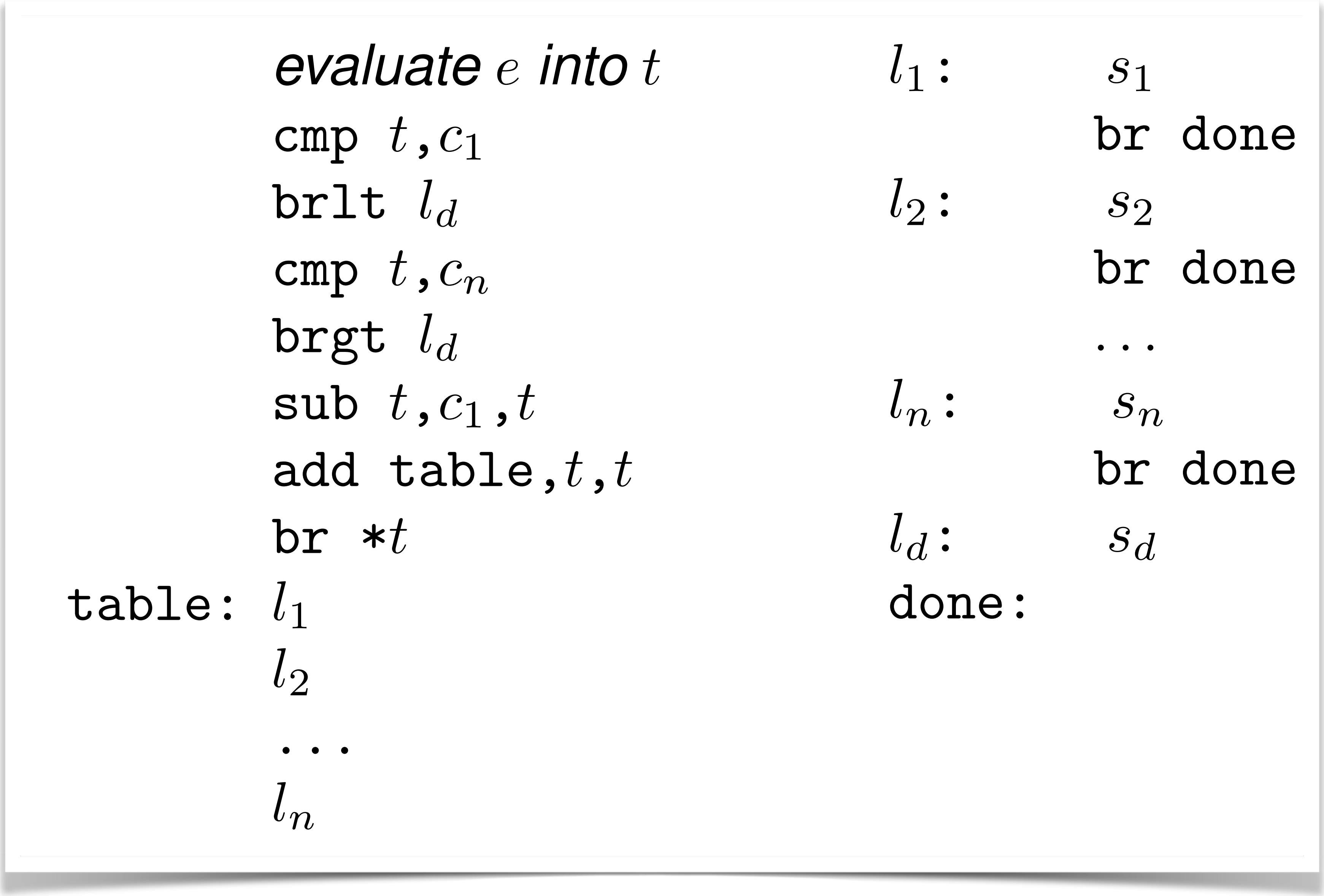
Dense `case` compilation

- If labels are dense in the range [c_1,c_n], it's better to use a jump table (O(1) time):



```
case $e$ of
  $c_1$ : $s_1$
  $c_2$ : $s_2$
  ...
  $c_n$ : $s_n$
  default : $s_d$
```

compiles to



```
        evaluate $e$ into $t$
        cmp $t$,$c_1$
        brlt $l_d$
        cmp $t$,$c_n$
        brgt $l_d$
        sub $t$,$c_1$,$t$
        add table,$t$,$t$
        br *$t$
table:  $l_1$
        $l_2$
        ...
        $l_n$

$l_1$:      $s_1$
            br done
$l_2$:      $s_2$
            br done
            ...
$l_n$:      $s_n$
            br done
$l_d$:      $s_d$
done:
```

Iteration: `while` and `repeat`

`while` e `do` s

compiles to

```
top:   evaluate e into t
       cmp t,true
       brneq done
       s
       br top
done:
```

`repeat` s `until` e

is equivalent to

```
s;
while not e do s
```

Counted loops

- Since iterating through a range of numbers is very common, many languages offer a dedicated statement,e.g.

$$\texttt{for}\ i\ \texttt{:=}\ e_1\ \texttt{to}\ e_2\ \texttt{do}\ s$$

- The detailed semantics vary, and can be tricky (e.g. can *s* change *i* ?)

- C/C++/Java offer a more general-purpose statement

$$\texttt{for}\ (e_1\texttt{;}\ e_2\texttt{;}\ e_3)\ s\texttt{;}$$

is equivalent to

$$e_1\texttt{;}\ \texttt{while}\ (e_2)\ \{\ s\texttt{;}\ e_3\ \}$$

Data-driven Iteration

- Many modern languages support generalized `for` loops that can iterate through any collection

```
val s = List(1,3,42,7)
for (v <- s)
  print(v+1) // prints 2 4 43 8
```

Scala

- In some languages this is implemented using iterators -- data objects that keep a pointer ("cursor") into the collection that can be advanced one element at a time
- Code above is shorthand for this:

```
val iter = s.iterator
while (iter.hasNext())
  print(iter.next()+1)
```

almost Scala

Loop exits

It can be useful to break out of the middle of a loop

```
loop
   $s_1$;
   exitif $e$;
   $s_2$
end
```

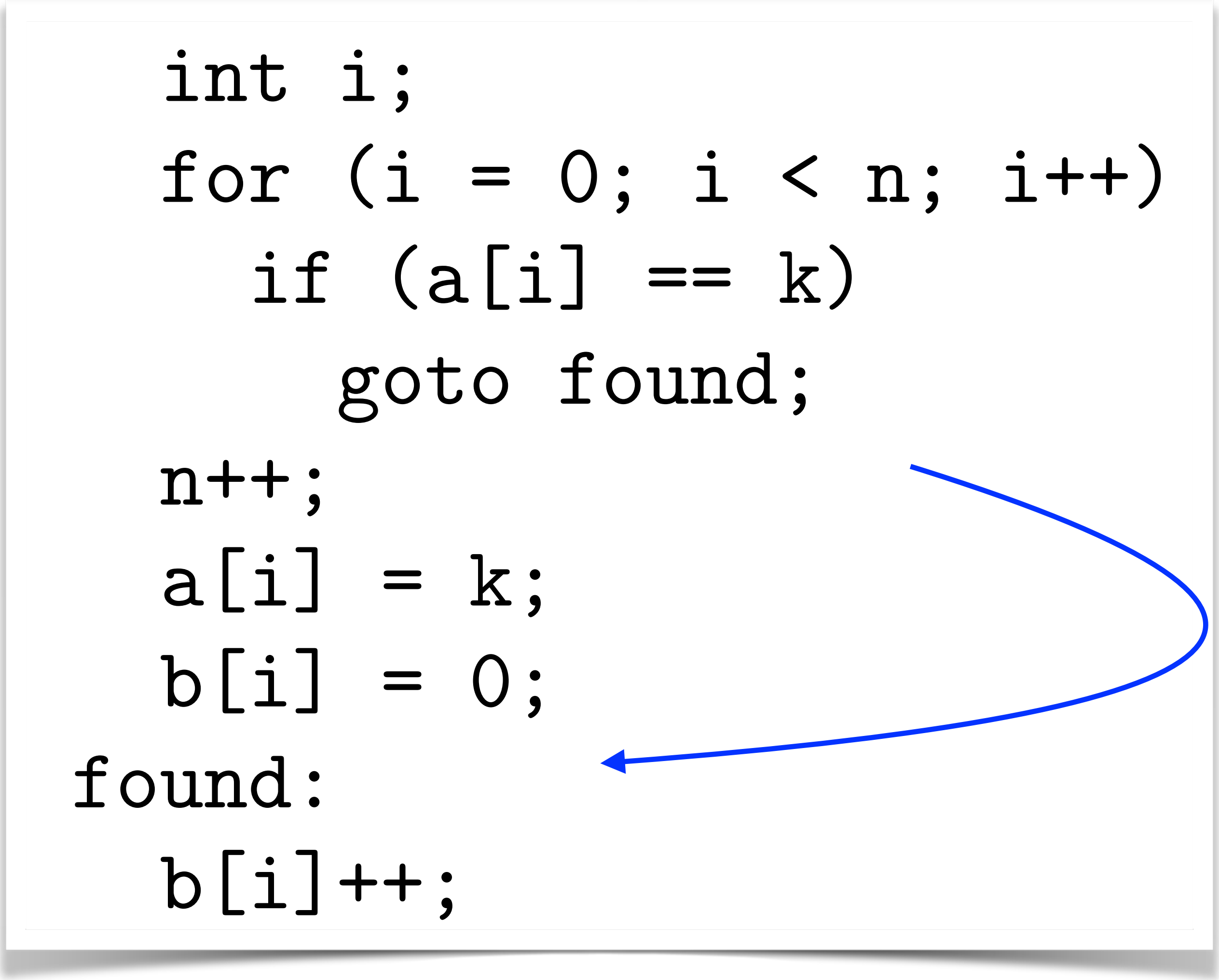
compiles to

```
top: $s_1$
     evaluate $e$ into $t$
     cmp $t$,true
     breq done
     $s_2$
     br top
done:
```

C/C++/Java `break` is unconditional form of `exit`
These languages also have a `continue` statement that jumps back to the top of the loop

Uses for goto ?

An efficient program using goto



```
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == k)
      goto found;
  n++;
  a[i] = k;
  b[i] = 0;
found:
  b[i]++;
```

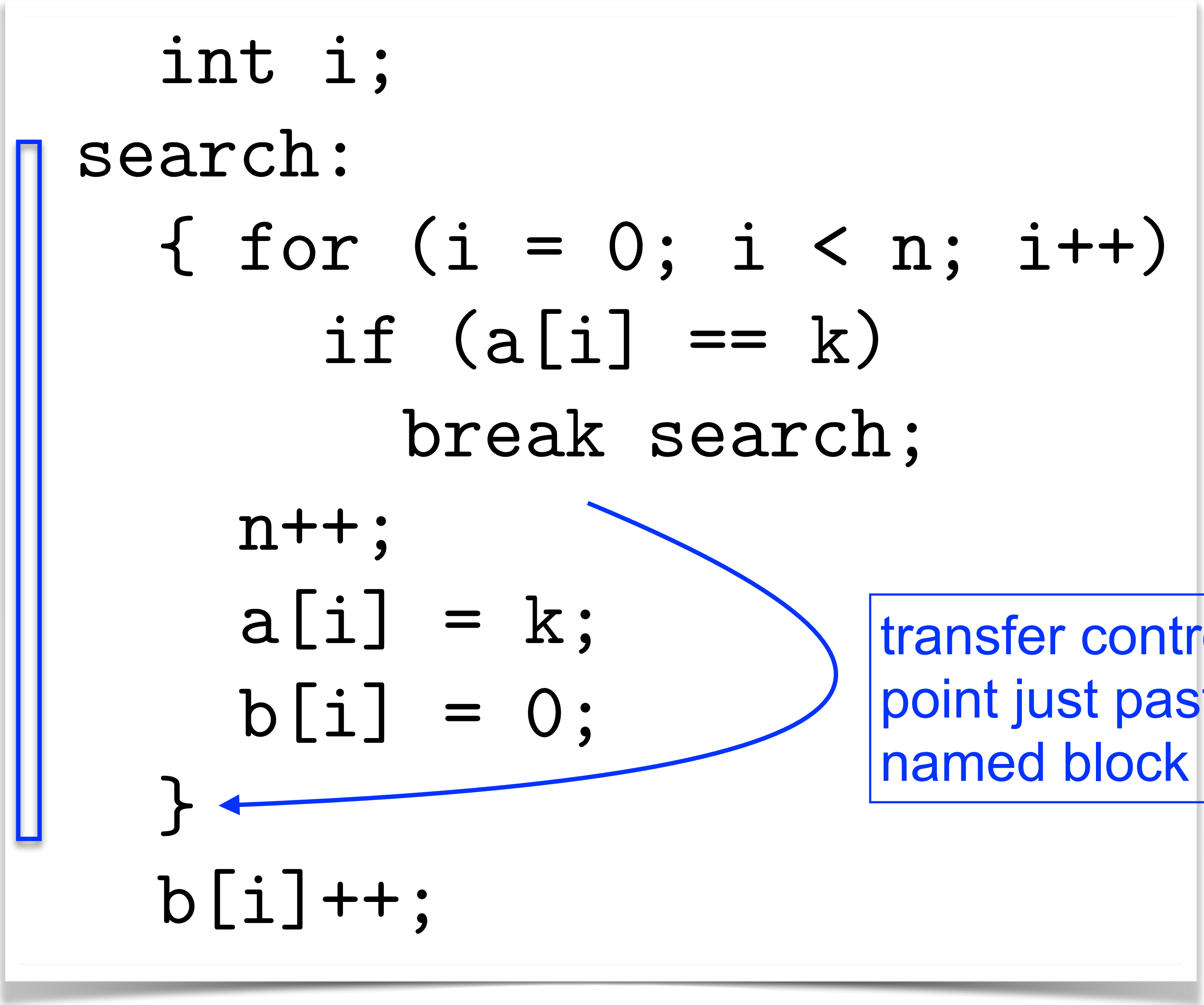
transfer control to
point well past end
of the loop

C

In most languages, there is no equivalently efficient program without goto: must add a flag variable

Multi-level `break`

But we can do as well in Java (or JavaScript, Go, ...), using a named, multi-level `break` statement



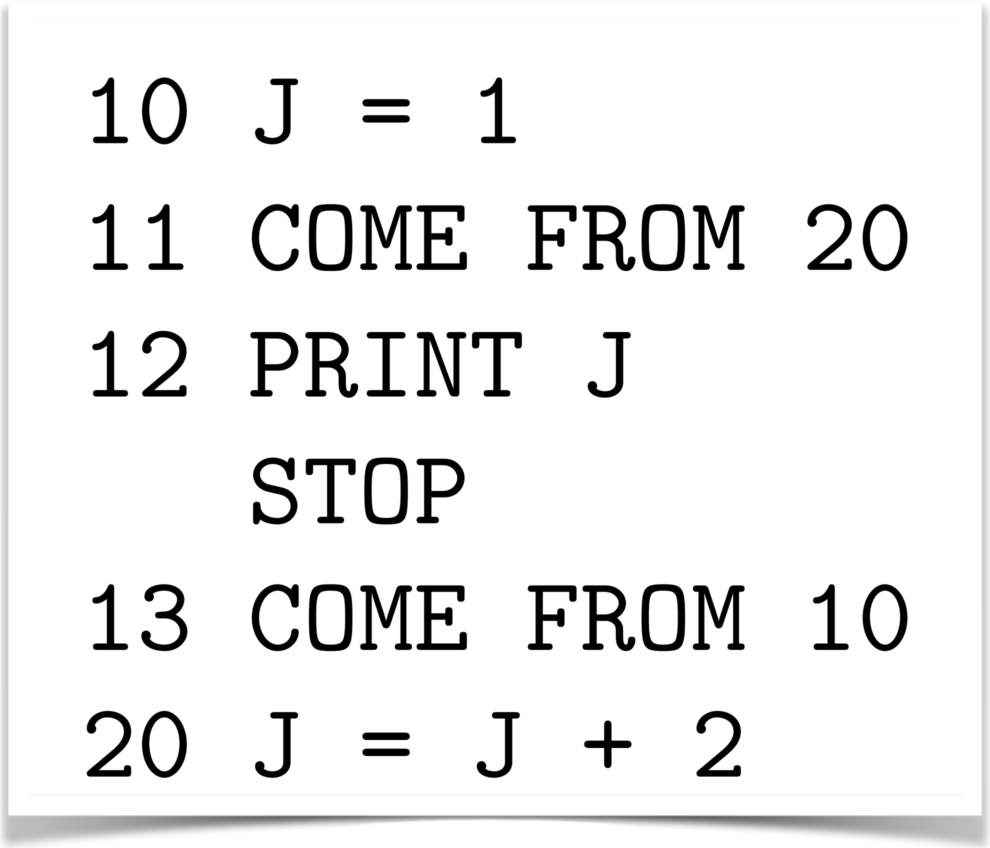
```
   int i;
search:
   { for (i = 0; i < n; i++)
         if (a[i] == k)
            break search;
     n++;
     a[i] = k;
     b[i] = 0;
   }
   b[i]++;
```

transfer control to point just past end of named block

Java

This construct was invented by Don Knuth in the 1960’s but not adopted into a mainstream language for 30 years!

The COME FROM statement



```
10 J = 1
11 COME FROM 20
12 PRINT J
   STOP
13 COME FROM 10
20 J = J + 2
```

R.Lawrence Clark, “A Linguistic contribution to GOTO-less programming,” *Datamation*, 19(2), 1973, 62-63.

- A notorious joke!
- But with a serious point: even with an ordinary GOTO, we must examine the whole label/branch structure of the program to understand its behavior