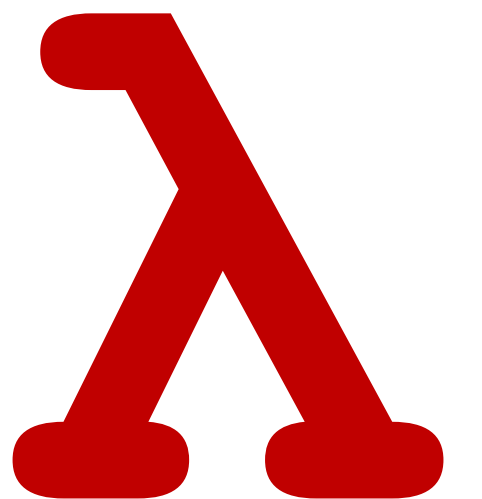


CSCI 275: Programming Abstractions

**Lecture 28: Control Flow Design
Spring 2025**

**Stephen Checkoway
Slides from Molly Q Feldman**



This Semester Thus Far

Thus far:

- First month we thought about how to *write* Racket
- Second month we thought about to *execute* Racket

The rest of the semester:

- Thinking about context *beyond* Racket (theory & practice)

Reminder: This Week's Goal

Talk about design of a language and how it impacts implementation

- In MiniScheme, you are implementing a certain language that has certain rules
- Many times, we have choices for these rules
- Last time & Today: what we *could* and *can* do for rules in language design
 - Another “instantiate your subconscious process” topic!
 - Another way to think about how your knowledge applies after this class

Language Design

Ways MiniScheme did not *deviate* from Racket

We decided to include control flow via:

- If-then-else statements
- Recursive evaluation of procedural approaches

Language structures that allow us to make choices about what statement happens next

Which of the following **control flow statements** are *not* part of the MiniScheme language?

- A. for loops
- B. while loops
- C. if statements
- D. cond statements
- E. More than one of the above

Ways MiniScheme did not *deviate* from Racket

We decided to include control flow via:

- If-then-else statements
- Recursive evaluation of procedural approaches

We did not consider other types of iteration or control flow constructs such as:

- for loops
- while loops
- (switch/match statements)

Why did we not consider other control flow?

Why did we not consider other control flow?

- If-then-else statements are *fundamental* in most languages
- Iteration via recursion is fundamental *to Racket* and, more broadly, to functional programming overall
 - Also an added benefit of reducing the need for additional special forms!
- These are also “standard” design constructs, that we see in many many languages

A (Very Different) Language Construct

Reminder from last time: Scope of a declaration

The *scope* of a declaration is the portion of the expression or program to which that declaration applies

Lexical binding

- Scope of a variable is determined by textual layout of the program
- C, Java, Scheme/Racket use lexical binding

Dynamic binding

- Scope of a variable is determined by most recent *runtime* declaration
- Bash and classic Lisp use dynamic binding

goto Statements

goto statements are a (classic) way to handle control flow in some programming languages

goto statements rely on two parts:

1. Add labels that reference specific code segments
2. Use goto label to move between code segments

This is C++ code.
What does it print
out?

A. 0 1 2 3 ... 9

B. 9 8 7 6 ... 0

C. 0 1 2 3 ... 10

D. Infinite sequence of 0s

E. Something else

```
1  #include <iostream>
2
3  int main()
4  {
5      int val = 0;
6      repeat:
7          if (val < 10) {
8              std::cout << val << " ";
9              goto repeat;
10         }
11     }
```

std::cout is like System.out.println
in Java

Does this change to the code solve the problem?

A. Yes

B. No

C. In some cases

```
1  #include <iostream>
2
3  int main()
4  {
5      int val = 0;
6      repeat:
7          if (val < 10) {
8              std::cout << val << " ";
9              val = val + 1;
10             goto repeat;
11         }
12 }
```

Introducing Complexity

```
1  #include <iostream>
2
3  int main()
4  {
5      int val = 0;
6      repeat:
7          if (val < 10) {
8              std::cout << val << " ";
9              val = val + 1;
10             goto repeat;
11         }
12 }
```

This example seems like “another way to iterate”

`goto` can introduce interesting consequences - especially for scope!

Languages with goto

Languages with `goto`:

- APL
- Ada
- Fortran
- Perl
- Assembly (you build if/for/while out of conditional gotos!)
- **C**
- **C++**

A Bit of Context: Objects in C++

```
class ObjectD {  
    public:  
        char val;  
        //constructor  
        ObjectD(char v) { val = v; };  
        // non-trivial destructor  
        ~ObjectD() { std::cout << val << ":d! "; }  
};
```

- Destructors start with ~ in C++
- Destructors called whenever an object is going to be destroyed
- Happens when they are called **explicitly** or **object goes out of scope**

Walk through an example!

Goal: how is this different than code
you've walked through before?

```
#include <iostream>

//In Class Goto Example
//Adapted from https://en.cppreference.com/w/cpp/language/goto
```

```
class ObjectD {
    char val;
public:
    //constructor
    ObjectD(char v) {val = v;};
    // non-trivial destructor
    ~ObjectD() {std::cout << val << ":d! "; }
};
```

```
int main() {

    int a = 10;

    std::cout << "before label" << "\n";

label:
    if (a == 10) {
        ObjectD obj = ObjectD('a');
    }
    else {
        ObjectD obj = ObjectD('b');
    }
    std::cout << a << " ";
    a = a - 2;

    if (a != 0) {
        goto label;
    }

    std::cout << "\n";
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            std::cout << "(" << x << "|" << y << ")" << "\n";
            if (x + y >= 3) {
                goto endloop;
            }
        }
    }

endloop:
    std::cout << "end loop" << "\n";
    goto label3;

label3:
    std::cout << "label3" << "\n";
}
```

goto examples adapted from the C/C++ guides:

<https://en.cppreference.com/w/c/language/goto>

<https://en.cppreference.com/w/cpp/language/goto>

Why is this not something we tend to use?

Why are `goto` statements not common in other languages?

Why did we not implement this in MiniScheme?

How did the *community* decide this?

- In conversation
- Overtime
- Due to real world challenges / challenging use cases

Goto Considered Harmful (1968)

Edsger Dijkstra wrote a letter to the editor as part of the Communications of the ACM

This letter is very well-known by academics (for instance, part of “Great Works” reading groups)

You might ask: why?

NOTE: This is a paper from 1968, the terminology &
approach is not modern.

<https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf>

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all “higher level” programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.