Computer Systems

Lecture 11

Records and Pointers

Dr José Cano, Dr Lito Michala
School of Computing Science
University of Glasgow
Spring 2020

Outline

- Compilation patterns
- Records
- Pointers
- Requests to the Operating System

Compilation patterns

- We have looked at several high level programming constructs
  - if b then S
  - if b then S else S
  - while b do S
  - for var := exp to exp do S

- There is a standard way to translate each to low level form
  - assignment
  - goto L
  - if b then goto L

- The low level statements correspond closely to instructions

Follow the patterns!

- Helps you understand precisely what high level language constructs mean (one of the aims of the course)
- It is essentially how real compilers work (another aim of the course)
- Saves time because
  - It's quicker to catch errors at the highest level (e.g. translating if-then-else to goto) rather than the lowest level (instructions)
  - It makes the program more readable, thus faster to check and debug
- Leads to good comments that make the program more readable
- The approach scales up to large programs
- Experienced programmers recognise the patterns, so if you use them your code is easier to read, debug and maintain

How can you know if you're using the patterns?

- Each pattern contains
  - **Changeable parts**: boolean and integer expressions, statements
  - **Fixed parts**: goto, if-then-goto
  - **Labels**: have to be different every time, but the structure of the fixed parts never changes
- Example: translating a while loop
  - **Start of the loop**: one comparison and one conditional jump
  - **End of the loop**: one unconditional jump

Are you using the pattern?

High level code

```
while bexp do
        S
```

The pattern for translation to low level

```
label1
        if bexp = False then goto label2
        S
        goto label1
label2
```

- The blue text is fixed (except you need to use unique labels)
- There should be one comparison, one conditional jump at the start of the loop
- There should be one unconditional jump at the end of the loop

Can you gain efficiency by violating the pattern?

- **Example**: avoid the cost of jumping to a test by transforming the while loop

```
label1
        if bexp = False then goto label2
label3
        S
        if bexp = True then goto label3
label2
```

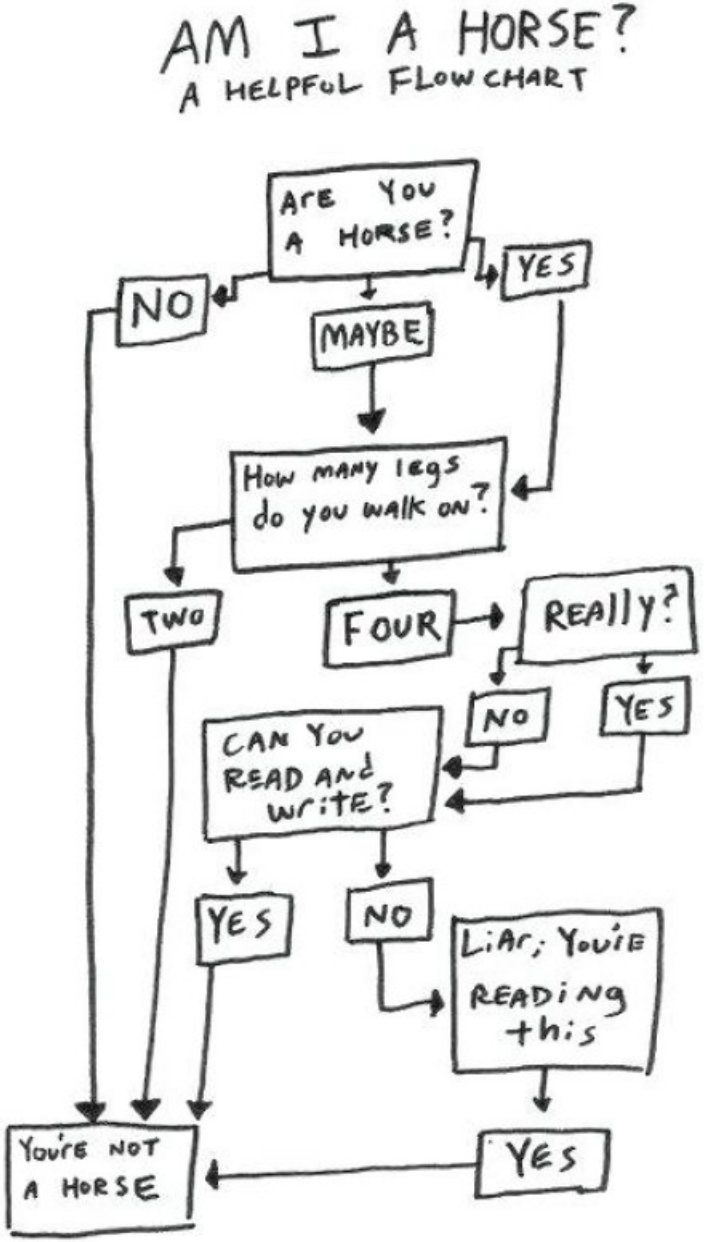
- If the loop executes a million times it saves at most one jump instruction!
- The code is longer, which can make it slower (e.g. because of cache)
- And when you do this in a large program it becomes incomprehensible
- Aim for readability and correctness

Comments

- Initial comments to identify the program, author, date, etc
- Early comments to say what the program does
- High level algorithm (in comments)
- Translation to low level algorithm (in comments)
- Translation to assembly language (with comments)
  - Every low level statement appears as a comment in assembly code
- Write the comments first!
  - The program development methodology entails writing the comments first
  - Avoid writing code first until it appears to work and then adding comments

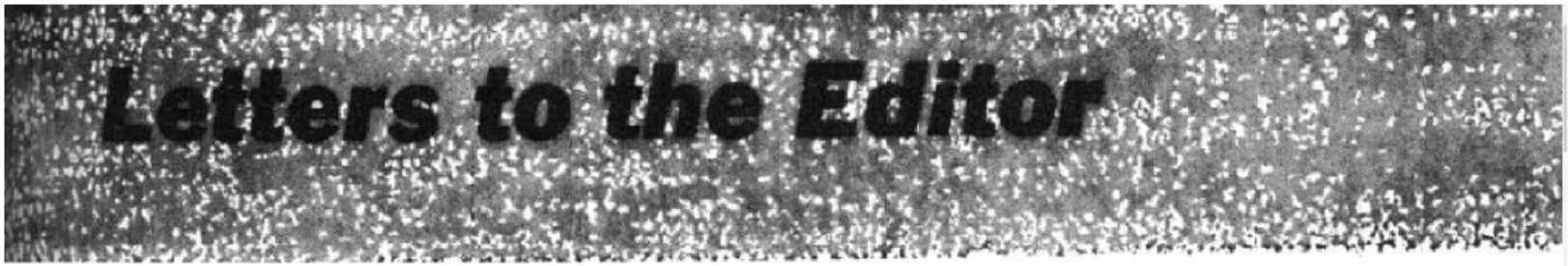
Why is goto controversial?

- If you develop code randomly, with goto jumping all over the place the program is hard to understand, unlikely to work, and difficult to debug
- This has given the goto statement a bad reputation
- But goto is essential for compilers because it's essentially the jump instruction
- The compilation patterns provide a safe and systematic way to introduce goto into a program
- But if you ignore the patterns, you lose these advantages
- Unstructured goto leads to complicated code


AM I A HORSE?
A HELPFUL FLOWCHART
ARE YOU A HORSE?
NO
MAYBE
YES
HOW MANY LEGS DO YOU WALK ON?
TWO
FOUR
REALLY?
NO
YES
CAN YOU READ AND WRITE?
YES
NO
LIAR; YOU'RE READING THIS
YES
YOU'RE NOT A HORSE

Edsger Dijkstra

Goto considered harmful: CACM 11(3), 1968



Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

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

dynamic progress is only characterized
call of the procedure we refer. With
we can characterize the progress of th
textual indices, the length of this se
dynamic depth of procedure calling.

Let us now consider repetition claus
or **repeat** *A* **until** *B*). Logically spea
superfluous, because we can express
recursive procedures. For reasons of
clude them: on the one hand, repeti
mented quite comfortably with prese
the other hand, the reasoning patte
makes us well equipped to retain ou
processes generated by repetition cla
the repetition clauses textual indices
describe the dynamic progress of the p
a repetition clause, however, we can
namic index," inexorably counting
corresponding current repetition. As
procedure calls) may be applied nest
progress of the process can always be

What happened next?

- Considered harmful
  - Dozens (hundreds?) of X considered harmful essays
- Goto elimination
  - **Theorem**: every program using goto can be expressed without goto, using while and if-then-else
- Structured programming
  - Programming paradigm aimed at improving the clarity, quality, and development time of a computer program
  - Effective way to develop programs by making extensive use of the structured control flow constructs (if-then-else, for, while, etc)

goto

https://xkcd.com/292/

Outline

- Compilation patterns
- Records
- Pointers
- Requests to the Operating System

Records

- A record contains several fields: access a field with the dot (.) operator

```
;   x, y :
;            record
;                { fieldA : int;
;                  fieldB : int;
;                  fieldC : int;
;                }
;
;   x.fieldA := x.fieldB + x.fieldC;
;   y.fieldA := y.fieldB + y.fieldC;
```

- Some programming languages call it a tuple or struct

Defining some records

```
; Data definitions

; The record x
x
x_fieldA data 3          ; offset 0 from x      &x_fieldA = &x
x_fieldB data 4          ; offset 1 from x      &x_fieldB = &x + 1
x_fieldC data 5          ; offset 2 from x      &x_fieldC = &x + 2

; The record y
y
y_fieldA data 20         ; offset 0 from y      &y_fieldA = &y
y_fieldB data 21         ; offset 1 from y      &y_fieldB = &y + 1
y_fieldC data 22         ; offset 2 from y      &y_fieldC = &y + 2
```

Naming each field explicitly

```
; ------------------------------------------------------
; Simplistic approach, with every field of every record named explicitly

; In record x, fieldA := fieldB + fieldC

; x.fieldA := x.fieldB + x.fieldC
load R1,x_fieldB[R0]
load R2,x_fieldC[R0]
add R1,R1,R2
store R1,x_fieldA[R0]
```

- There's a better way!

Outline

- Compilation patterns
- Records
- Pointers
- Requests to the Operating System

Pointers

- So far, we have been finding a piece of data by giving it a label

  load R2,xyz[R0]

  ...

  xyz data 5

- An alternative way to find the data is to make a pointer to it
  - A pointer is an address

- **&x** means the address of x (a pointer to x): you can apply the & operator to a variable but not to a complex expression
  - &x is ok
  - &(3*x) is not ok

- ***p** means the value that p points to: you can apply the * operator to any pointer

Expressions using pointers

- **The & operator** gives the address of its operand
  - p := x puts the value of x into p
  - p := &x puts the address of x into p (the address of x is called a pointer to x, and we say “p points at x”)

- **The * operator** follows a pointer and gives whatever it points to
  - *p is an expression whose value is whatever p points to
  - y := p stores p into y, so y is also now a pointer to x
  - y := *p follows the pointer p, gets the value (which is x) and stores it in y

The & operator requires only one instruction: lea!

```
lea R5,x[R0]          ; R5 := &x
...
lea R6,y[R0]          ; R6 := &y
store R6,p[R0]        ; p := &y
...
x data 25
y data 0
p data 0
```

The * operator requires only one instruction: load!

```
load R7,p[R0]          ; R7 := p
load R8,0[R7]          ; R8 := *p
```

Flexibility of load and lea

- We have now seen two ways to use lea
  - To load a constant into a register: `lea R1,42[R0]` ; R1 := 42
  - To create a pointer: `lea R2,x[R0]` ; R2 := &x
  - lea can do more…

- And there are several ways to use load
  - To load a variable into a register: `load R3,x[R0]` ; R3 := x
  - To access an array element: `load R4,a[R5]` ; R4 := a[R5]
  - To follow a pointer: `load R6,0[R7]` ; R6 := *R7

Following a pointer to the address of x gives x

- The value of *(&x) is just x!

```
lea R4,x[R0]        ; R4 := &x
load R5,0[R4]       ; R5 := *(&x) = x

load R6,x[R0]       ; R6 := x
```

Review: accessing a variable the ordinary way

- Low level language (same as high level)

```
x := x + 5;
```

- Assembly language

```
; Accessing variable x by its address, with R0
lea R1,5[R0]          ; R1 := 5 (constant)
load R2,x[R0]         ; R2 := x
add R2,R2,R1          ; R2 := x + 5
store R2,x[R0]        ; x := x + 5
```

Accessing a variable through a pointer

- Low level language (same as high level)

```
x := x + 5;
```

- Assembly language

```
; Put a pointer to x into R3, which contains the address of x
; R3 := &x
lea R3,x[R0]            ; R3 := &x

; Add 5 to whatever word R3 points to
lea R1,5[R0]            ; R1 := 5 (constant)
load R4,0[R3]           ; R4 := *R3
add R4,R4,R1            ; R4 := *R3 + 5
store R4,0[R3]          ; *R3 := *R3 + 5
```

- Equivalent to *(&x) := *(&x) + 5

Why are pointers helpful?

- We can write a block of code that accesses variables through pointers
  - Example: a record
- We can reuse it by executing it with the pointer set to point to different data
  - Example: an array of records
- Later, we'll see additional benefits of using pointers…

Access a record using a pointer

```
; Make the same code work for any record with the same fields
; Set x as the current record by making R3 point to it
lea R3,x[R0]              ; R3 := &x

; Perform the calculation on the record that R3 points to
load R1,1[R3]             ; R1 := (*R3).fieldB
load R2,2[R3]             ; R2 := (*R3).fieldC
add R1,R1,R2              ; R1 := (*R3).fieldB + (*R3).fieldC
store R1,0[R3]            ; *R3.fieldA := (*R3).fieldB + (*R3).fieldC
```

Outline

- Compilation patterns
- Records
- Pointers
- Requests to the Operating System

Requests to the Operating System

- Many operations cannot be performed directly by a user program because
  - **Main reason**: user could violate system security
  - Some operations are difficult to program
  - The code would need to change when OS is updated

- The program requests the operating system to perform them
  - Executing a trap instruction, for example trap R1,R2,R3

- A trap is a jump to the Operating System
  - And you don't have to give the address to jump to

- We use pointers to tell the operating system what to do

Typical OS requests

- The type of request is a number, placed in R1, and operands (if any) are in R2, R3

- The specific codes used to make a request are defined by the operating system, not by the hardware
  - This is a major reason why compiled programs run only on one operating system

- Typical requests
  - Terminate execution of the program
  - Read from a file
  - Write to a file
  - Allocate a block of memory

Termination

- A program cannot stop the machine
  - It requests the operating system to terminate it

- The operating system then removes the program from its tables of running programs, and reclaims any resources dedicated to the program

- In Sigma16, you request termination by trap R0,R0,R0

Character strings: pointer to array of characters

- A string like “The cat in the hat” is represented as an array of characters
- Each element of the array contains one character
- If you are writing a string to output, the last character of the string should be a “newline character”

Write operation on Sigma16

- To write a string of characters
  - trap R1,R2,R3
  - R1: 2 is the code that indicates a write request
  - R2: address of first character of string to write
  - R3: length of string (the last word should be newline character)
  - See example program Write.asm.txt

Writing a string

- To write a string named out, we use (1) lea to load a constant, (2) lea to load the address of an array, (3) load to get a variable

```
; write out (size = k)
      lea R1,2[R0]          ; trap code: write
      lea R2,animal[R0]     ; address of string to print
      load R3,k[R0]         ; string size = k
      trap R1,R2,R3         ; write out (size = k)
      trap R0,R0,R0         ; terminate
k data 4                    ; length of animal
; animal = string "cat"
animal
      data 99               ; character code for 'c'
      data 97               ; character code for 'a'
      data 116              ; character code for 't'
      data 10               ; character code for newline
```

pointers

https://xkcd.com/138/