Computer Systems

Lecture 8

Control Structures

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

Copyright ©2020 John T. O'Donnell

Outline

- History: the first programmer in the world
- Programming languages and Compiling
- Jumping and comparing
- Compilation patterns
- Programming techniques

History: the first programmer in the world

- Who could it be?

- There was one person who…
  - Wrote the first substantial programs for a real computer
  - Made fundamental discoveries that underlie programming

- This work also contained ideas related to the discoveries of Alan Turing

Ada Lovelace (1815-1852)

- Daughter of Lord Byron, the poet
- Deeply interested in science, mathematics, and technology
- Studied Babbage's plans for Analytical Engine
- Worked out something that Babbage hadn't fully realised
  - The Engine could store programs in its memory
- Even more significantly, she also realised the profound significance of this
- A major programming language Ada was named after her

Translation of monograph on Analytical Engine

- Luigi Menabrea, an Italian engineer, visited Babbage and wrote a monograph on the Analytical Engine

- Ada translated it into English

- She found the text sketchy, and extended it with “Notes”
  - It's worth reading!
  - http://www.fourmilab.ch/babbage/sketch.html

- The Notes contained original research and completely new insights

Ada Lovelace

"The world's first computer programmer"

Outline

- History: the first programmer in the world
- Programming languages and Compiling
- Jumping and comparing
- Compilation patterns
- Programming techniques

Compiling

- Computers cannot execute programs in a high-level programming language
- We must translate a program into assembly language
- Translating from high-level language to assembly language is called compiling
- This is done by software called a compiler: it reads in a program in e.g. C++ and translates it to assembly language
- Benefits of using compilers
  - A computer can run programs in many languages (using many compilers)
  - Compilers can make programming easier: good error messages, etc
  - Languages can be designed to fit well for different purposes
- For each type of high level language construct, we will translate to assembly language following a standard pattern

Statements in high level languages

- A program contains
  - Statements that perform calculations
    - Assignment statements (e.g. x := 2 + 3)
  - Statements that determine the order of the calculations (control structures)
    - **Conditionals**: if-then-else
    - **Loops**: while, repeat, for
    - **Structuring computation**: functions, procedures, coroutines, etc

High level control structures

- Notation
  - S, S1, S2, etc means “any statement” (e.g. an assignment statement)
  - bexp means “any boolean expression”, either True or False (e.g. x>3)

- **Block**: We can treat several consecutive statements as just a single statement {S1; S2; S3;}

- **if-then**: if bexp then S;

- **if-then-else**: if bexp then S1 else S2;

- **while-loop**: while bexp do S

- And there are many more

Low level constructs

- **Assignment statements**: x := a * 2
- **Goto**: goto computeTotal
- **Conditional**: if x < y then goto loop
- First translate high-level constructs into these low-level statements
- Then translate the low-level statements into assembly language

The Goto statement

```
      S;
loop: S;
      S;
      S;
      goto loop;
```

- Many (not all) programming languages have a goto statement
- Any statement may have a label (for example “loop”)
- Normally execution proceeds from one statement to the next, on and on
- A goto L transfers control to the statement with label L

Using the goto statement

- The first programming language (Fortran, 1955) didn't have fancy control structures
  - You had to do nearly everything with goto
- But goto leads to unreadable programs and unreliable software
- The modern view
  - In a high level language, you should not use goto
  - For low level programming (like assembly language) the goto serves as the foundation for implementing the higher level control statements
- We will use two forms
  - Inconditional: goto L
  - Conditional: if b then goto L

The conditional goto statement

- if bexp then goto label
- bexp is a Boolean expression: x < y, j = k, abc > def
- If the bexp is True the statement goes to the label
- Otherwise we just move on the the next statement
- The only thing you can put after then is a goto statement

Outline

- History: the first programmer in the world
- Programming languages and Compiling
- Jumping and comparing
- Compilation patterns
- Programming techniques

Machine language: Jumping

- The foundation of control structures is jump instructions
- Jumping is the machine language equivalent of goto
- An instruction may have a label
- The label is a name, starting with a letter, and must appear starting in the first character of a line
- The unconditional instruction jump loop[R0] means goto loop

Comparison instruction: Boolean form

- cmplt R2,R5,R8
- Means “compare for Less Than”
- The operands are compared: R5 < R8
- This gives a Boolean, 0 (for False) or 1 (for True)
- That Boolean result is loaded into the destination R2
- There are three of these instructions
  - cmplt: compare for Less Than
  - cmpeq: compare for Equal
  - cmpgt: compare for Greater Than

Conditional jumps: Boolean decision

- There are two instructions: you can jump if a Boolean is False or True

- **jumpf**: jump if False
  - jumpf R4,label1[R0]
  - Means if R4 contains False, then goto label1
  - 0 means False, so this means if R4=0 then goto label1

- **jumpt**: jump if True
  - jumpt R5,label2[R0]
  - Means if R5 contains True, then goto label2
  - Any number other than 0 means True, so this means if R5 $\neq$ 0 then goto label2

Outline

- History: the first programmer in the world
- Programming languages and Compiling
- Jumping and comparing
- Compilation patterns
- Programming techniques

Compilation patterns

- Each programming construct is translated according to a standard pattern

- It's useful to translate in two steps
  - First, translate complex statements to simple low level statements
    - go to label, if b then goto label
    - The “goto” form corresponds closely to machine instructions
  - Then complete the translation to assembly language
    - Assignment statements: loads, then arithmetic, then store
    - goto label: jump label[R0]
    - if b then goto label: jumpt R5,label[R0] where R5 contains b
    - if not b then goto label: jumpf R5,label[R0] where R5 contains b

Compiling an assignment statement

- Load the operands; do calculations; store results

```
; x := a + b*c;
    load R1,a[R0]        ; R1 = a
    load R2,b[R0]        ; R2 = b
    load R3,c[R0]        ; R3 = c
    mul R4,R2,R3         ; R4 = b*c
    add R4,R1,R4         ; R4 = a + (b*c)
    store R4,x[R0]       ; x := a+(b*c)
```

if bexp then S

```
if x<y
    then {statement 1;}
statement 2;
```

Translates into

```
    R7 := (x < y)
    jumpf R7,skip[R0]
    instructions for statement 1
skip
    instructions for statement 2
```

Example: code with if-then

- Source program fragment

```
x := 2;
if y>x
        then { a := 5; }
b := 6;
```

Example: translating if-then

```
; x := 2;
      lea R1,2[R0]            ; R1 := 2
      store R1,x[R0]          ; x := 2

; if y>x
      load R1,y[R0]           ; R1 := y
      load R2,x[R0]           ; R2 := x
      cmpgt R3,R1,R2          ; R3 := (y>x)
      jumpf R3,skip[R0]       ; if y <= x then goto skip

; then { a := 5; }
      lea R1,5[R0]            ; R1 := 5
      store R1,a[R0]          ; a := 5

; b := 6;
skip  lea R1,6[R0]            ; R1 := 6
      store R1,b[R0]          ; b := 6
```

if bexp then S1 else S2

```
if x<y
    then { S1 }
    else { S2 }
S3
```

Compiled into

```
    R5 := (x<y)
    jumpf R5,else[R0]
; then part of the statement
    instructions for S1
    jump done[R0]
; else part of the statement
else
    instructions for S2
done
    instructions for statement S3
```

while b do S

```
while i<n do
    { S1 }
S2
```

Compiled into

```
loop
    R6 := (i<n)
    jumpf R6,done[R0]
    … instructions for the loop body S1
    jump loop[R0]
done
    instructions for S2
```

Infinite loops

```
while (true)
    {statements}
```

Compiled into

```
loop
    ... instructions for the loop body
jump loop[R0]
```

Nested statements

- For each kind of high level statement, there is a pattern for translating it to
  1. Low level code (goto)
  2. Assembly language

- In larger programs, there will be nested statements

```
if b1
     then { S1;
               if b2 then {S2} else {S3};
               S4;
           }
     else { S5;
               while b3 do {S6};
          }
S7
```

How to compile nested statements

- A block is a sequence of instructions where
  - To execute it, always start with the first statement
  - When it finishes, it always reaches the last statement
- Every high-level statement should be compiled into a block of code
- This block may contain internal structure (several smaller blocks)
  - To execute it you should always begin at the beginning and it should always finish at the end
- The previous patterns work for nested statements
- You need to use new labels (can't have a label like “skip” in several places)

Outline

- History: the first programmer in the world
- Programming languages and Compiling
- Jumping and comparing
- Compilation patterns
- Programming techniques

Programming techniques

- There are two ways to handle variables
  - The **statement-by-statement style** (use it if you feel confused)
    - Each statement is compiled independently
    - load, arithmetic, store
    - Straightforward but inefficient
  - The **register-variable style** (use it if you like the shorter code it produces)
    - Keep variables in registers across a group of statements
    - Don't need as many loads and stores
    - More efficient
    - You have to keep track of whether variables are in memory or a register
    - Use comments to show register usage
    - Real compilers use this style

Examples of the two styles

- Translate the following program to assembly language using each style

```
x = 50;
y = 2*z;
x = x+1+z;
```

Examples of statement-by-statement style

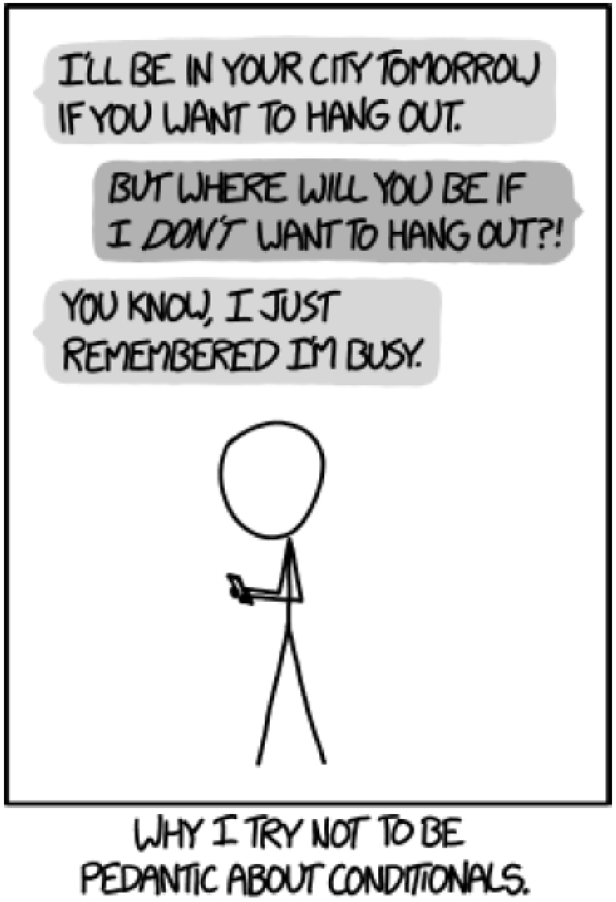
```
; x = 50;
    lea R1,$0032        ; R1 = 50
    store R1,x[R0]      ; x = 50

; y = 2*z;
    lea R1,$0002        ; R1 = 2
    load R2,z[R0]       ; R2 = z
    mul R3,R1,R2        ; R3 = 2*z
    store R3,y[R0]      ; y = 2*z

; x = x+1+z;
    load R1,x[R0]       ; R1 = x
    lea R2,1[R0]        ; R2 = 1
    load R3,z[R0]       ; R3 = z
    add R4,R1,R2        ; R4 = x+1
    add R4,R4,R3        ; R4 = x+1+z
    store R4,x[R0]      ; x = x+1+z
```

Example of register-variable style

```
; Usage of registers
; R1 = x
; R2 = y
; R3 = z
; x = 50;
    lea R1,$0032        ; x = 50
    load R3,z[R0]       ; R3 = z
    lea R4,$0002        ; R4 = 2
; y = 2*z;
    mul R2,R4,R3        ; y = 2*z
; x = x+1+z;
    lea R4,$0001        ; R4 = 1
    add R1,R1,R4        ; x = x+1
    add R1,R1,R3        ; x = x+z
    store R1,x[R0]      ; move x to memory
    store R2,y[R0]      ; move y to memory
```

Comparison of the two styles

- Statement by statement
  - Each statement is compiled into a separate block of code
  - Each statement requires loads, computation, then stores
  - A variable may appear in several different registers
  - There may be a lot of redundant loading and storing
  - Object code corresponds to source code, but it may be unnecessarily long
- Register variable
  - The instructions corresponding to the statements are mixed together
  - Some statements are executed entirely in the registers
  - A variable is kept in the same register across many statements
  - The use of loads and stores is minimised
  - Object code is concise but harder to see how it corresponds to source code
- It's possible a mixture of styles: no need to follow one or other all the time



https://xkcd.com/1652/