Computer Systems 1

Lecture 22

Retrospective and Looking Forward

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

1. Revision
   - Circuits
   - Architecture
   - Compilation patterns
2. Programming fundamentals
3. Looking forward!

Announcements

- Today is the last lecture!
- But there is an advanced topic lecture tommorow
  - A processor circuit: a digital circuit that fully implements a CPU, and you can run programs on it
  - 2pm, SAWB 303. Take lift to level 3, in the new building; it's the "triangle room" in the middle of level 3
- Tutors want to help you finish the assessed exercise
- Submit it even if it isn't finished or working
- Don't forget Quiz 10 (closes Friday) and Quiz 11 (closes Friday next week)

Revision

- The most important topics are the ones we have spent the most time on
- Read through all the materials
  - Lecture slides
  - Lab handouts and solutions
- Primary topics
  - Circuits
  - Architecture and assembly language
- Shorter topics
  - Number representation: binary, two's complement
  - Interrupts and processes

Circuits

- Primitive components
  - logic gates: inv and2 or2 xor2
  - clock: ticks, cycles, valid signals
  - flip flops: dff
  - Understand the most important circuits
    - multiplexer: mux1
    - register: reg1

Instructions

- You need to know what the basic instructions do and how to use them
  - Memory and addresses: load store lea
  - Arithmetic: add sub mul div
  - Comparison: cmplt cmpeq cmpgt
  - Jumps: jump jumpt jumpf jal
  - System: trap R0,R0,R0 (just for halting, not for I/O)
- Instruction representation
  - You should understand the concept
  - But you do not need to remember the details: the exam does not ask you to convert any instructions from assembly language to machine language

Addresses and data structures

- Effective addresses: sum of displacement and register
- Accessing an element of an array
- Accessing an element of a record
- Pointers: the (*p) and (&x) operators
- Linked list traversal

High and low level programming constructs

- High level
  - if then, if then else, case, while loops, for loops
- Low level
  - assignment, goto, if then goto

High and low level programming constructs

- The low level statements correspond to machine instructions
- Assignment statement
    - Load the operands in the expression into registers
    - Do the arithmetic
    - Store the result into a variable in memory
    - You can also keep variables in registers over a larger block of code
- goto label
    - jump label[R0]
- if b then goto label
    - Evaluate the boolean expression, put it in a register
    - conditional jump: either jumpt or jumpf

Compliation patterns

- Systematic pattern for translating each high level construct into low level statements
- Most high level constructs contain a Boolean expression
- Translate this into goto and if-then-goto statements that cause the right blocks of instructions to be executed
- Check that the translation is correct by hand executing with both values of the Boolean: True, False
- Case statements use a jump table

How do you learn programming

- The approach to learning programming has changed over the years
- First programming languages
  - Learn the statements and what they do
  - Statements are low level
- Large scale software
  - Software becoming complex
  - goto considered harmful
- Problem solving
  - Programming languages have complex statements, control structures and data structures
  - Teach "problem solving"
  - Use vague English and some examples to explain what the language constructs do

A hypothesis

- Project with Fionnuala Johnson, Stephen McQuistin, John O'Donnell
- Hypothesis
  - Programmers need *both* solid grasp of fundamentals *and* problem solving skills
  - The fundamentals include a model of what language constructs mean
  - Good models include
    - Translation from higher level to lower level (because the lower level constructs are simpler)
    - Diagrams showing the data structures and control flow: "box and arrow" diagrams, structure of the call stack, etc

Connections with other subjects

Similar debates occur in many subjects

- Natural language
  - A popular idea: learning grammar impedes creativity
  - Alternative view: knowing grammar enables the ability to express your ideas
- Arts and crafts and music
  - Should you learn how to use the tools of the trade?
  - Or just pick up how to use them while "expressing" yourself?

An experiment: surveys on programming fundamentals

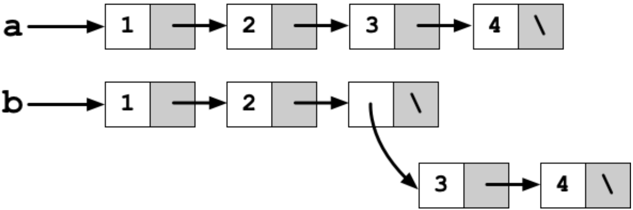
- Try some problems where misunderstandings of the fundamentals will lead to wrong answers
- See if we can provide models that improve the results
- Which language to use?
  - Ideally, we would use an algorithmic language (Algol family)
  - This would enable us to show in detail the translation from high level to low level
  - For practical reasons, we must start with a language most widely known by the students: in this case, Python
  - It's not feasible to show the precise translations from Python to low level (far too complex)
  - But we can at least point out some specifics
    - Show what list operations + append extend do
    - Show what break and continue really mean

Basic list operations: extend and append

```
a = [1, 2]
a.extend([3,4])
b = [1, 2]
b.append([3,4])
```

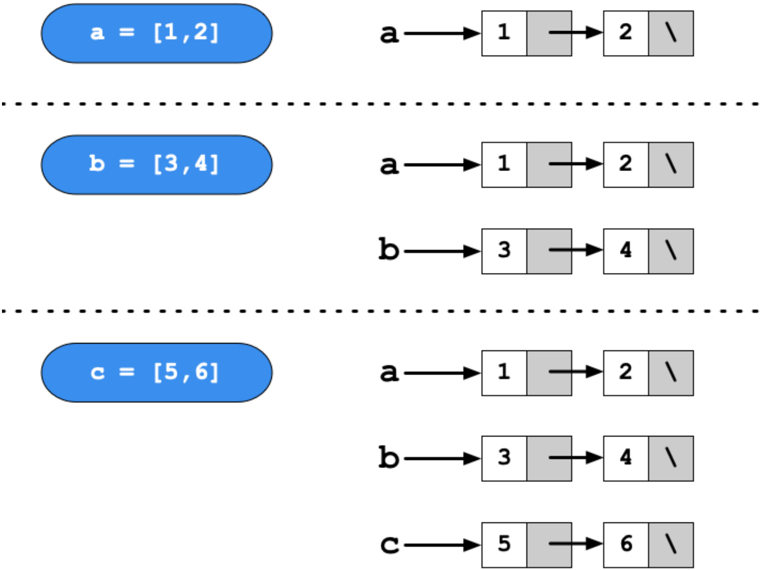
```
a =  [1, 2, 3, 4]
b =  [1, 2, [3, 4]]
```

Effect of extend and append on data structures

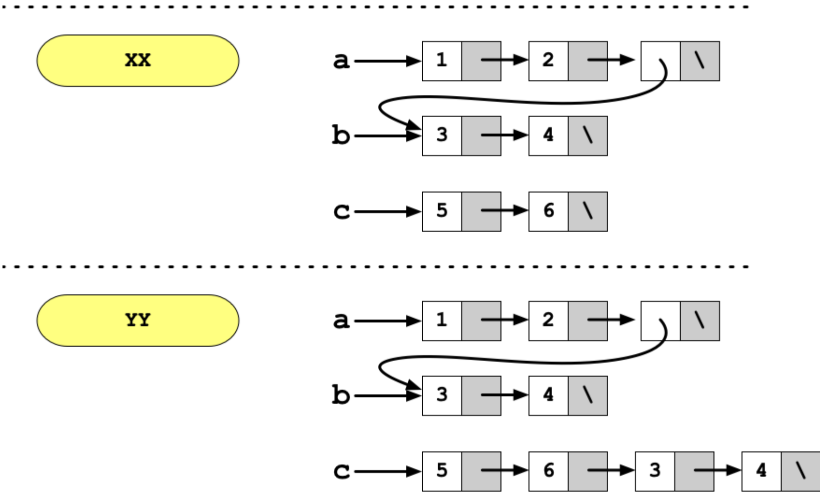


- Need to be able to read a “box and arrow” diagram and work out what the lists are
- a = [1, 2, 3, 4]
- b = [1, 2, [3, 4]]

Make some lists



XX changed a, YY changed c. Which is append, which extend?



List manipulation: abc

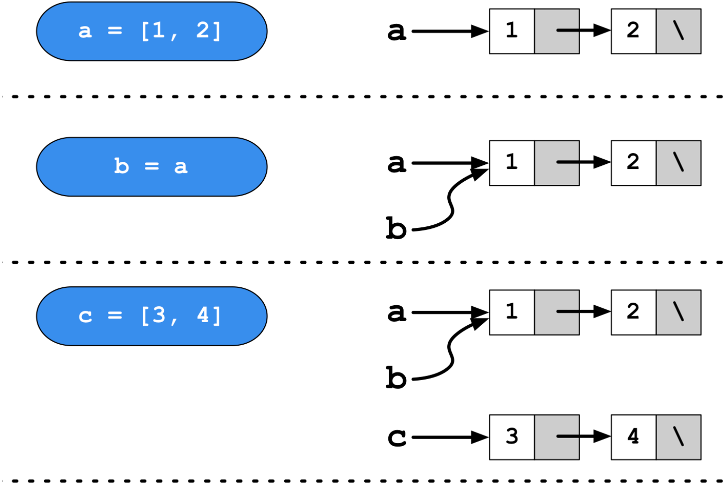
```
a = [1, 2]
b = a
c = [3, 4]
a = a + c
c.append(5)
```

```
a = [1, 2, 3, 4]
b = [1, 2]
c = [3, 4, 5]
```

Almost half answered [1,2,3,4]

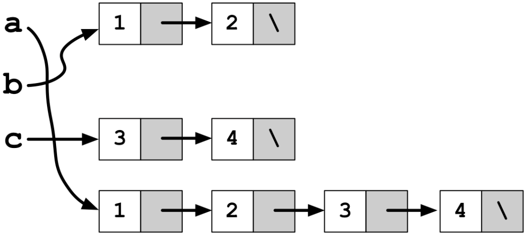
- **In a = a+c, the nodes in a are not changed. A new list is created and a is made to point to that**
- b still points to the nodes that comprised the *original* value of a
- Here the lists are mutable, but you could also use a = a + c on *immutable* data (like strings) because the + operator *does not change the data*, it just creates a *new* value

List manipulation abc: initial values

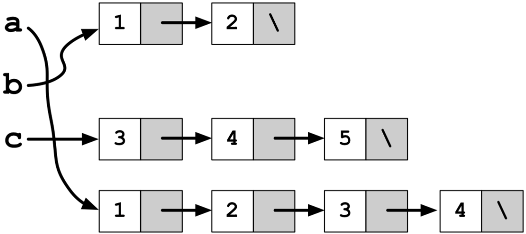


List manipulation abc: after + and append

`a = a + c`



`c.append(5)`



List manipulation: def

```
d = [1, 2]
e = d
f = [3, 4]
d.append(f)
f.append(5)
```

```
d =  [1, 2, [3, 4, 5]]
e =  [1, 2, [3, 4, 5]]
f =  [3, 4, 5]
```

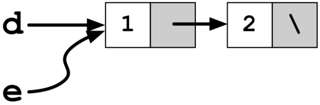
Almost half answered [1,2]

- append modifies the data structure, it doesn't produce a new list
- After the appends, e and f still point to the same nodes they did before, but those nodes now point to lists with changed data
- append can only be used on a mutable value such as a list, but not on an immutavle value such as a string

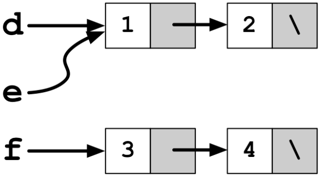
List manipulation: def initial values

`d = [1, 2]`

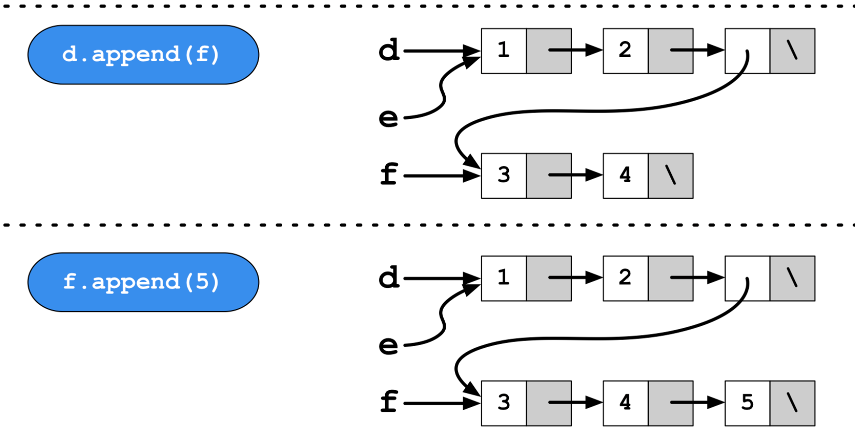
`e = d`



`f = [3, 4]`



List manipulation: def after appends



List manipulation: ghi

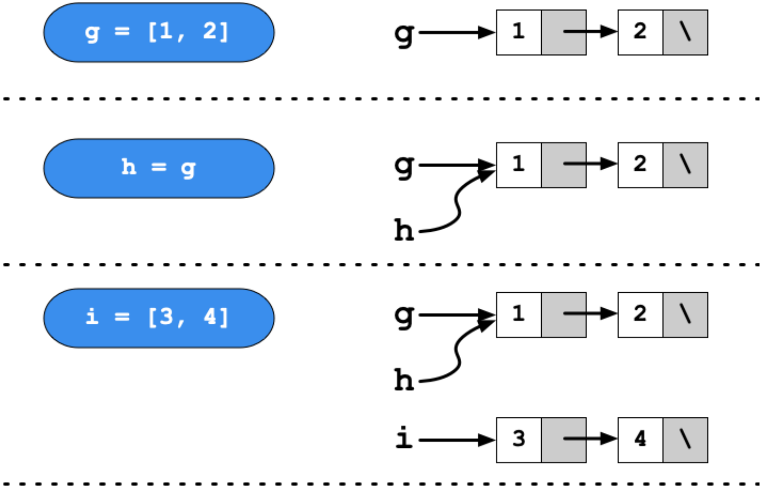
```
g = [1, 2]
h = g
i = [3, 4]
g += i
i.append(5)
```

```
g =  [1, 2, 3, 4]
h =  [1, 2, 3, 4]
i =  [3, 4, 5]
```

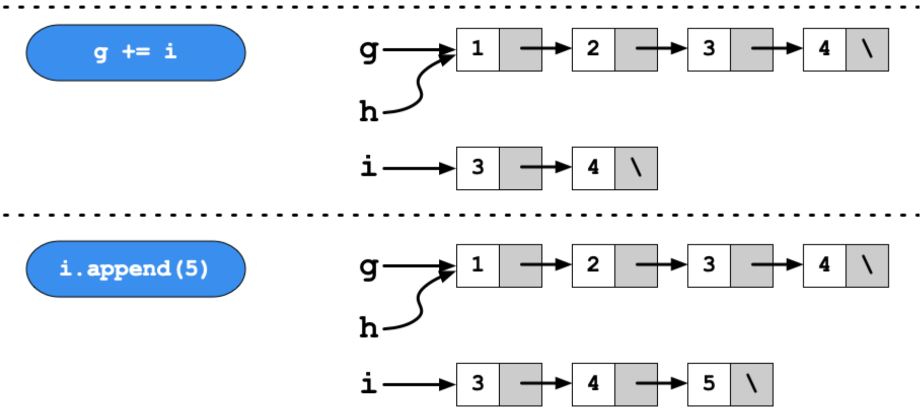
A majority answered [1,2]

- g += i modifies the representation of g (unlike g+i).
- g (and h) still point to the same node, but the list is changed
- The list that i points to is copied into the end of g, extending it, but these nodes are copies of the nodes in i
- i.append(5) modifies the representation of i, but not g (or h)

List manipulation: ghi initial values



List manipulation: ghi after += and append

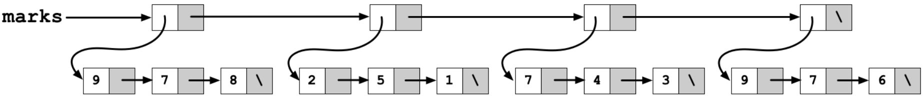


For loop

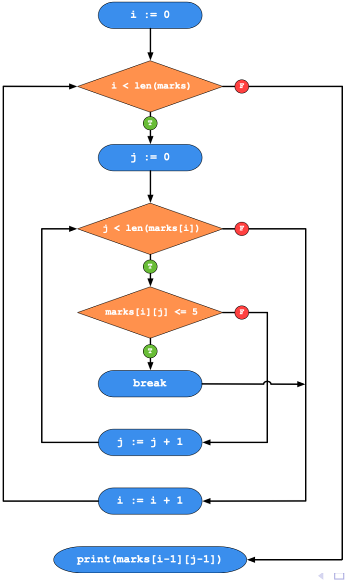
```
student_marks = [[9,7,8], [2,5,1], [7,4,3], [9,7,6]]
for student in student_marks:
    for mark in student:
        if mark <= 5:
            break
print ('mark = ', mark)
```

```
mark =  6
```

Data structure used in for loop



A flowchart

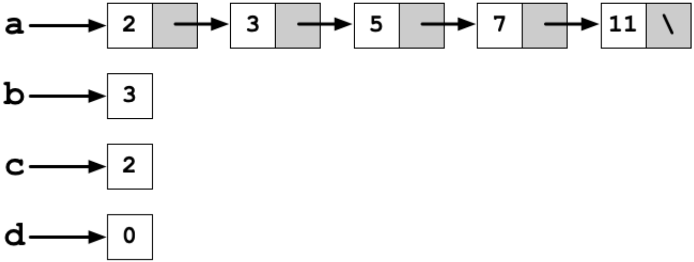

i := 0
i < len(marks)
F
T
j := 0
j < len(marks[i])
F
T
marks[i][j] <= 5
F
T
break
j := j + 1
i := i + 1
print(marks[i-1][j-1])

If statement

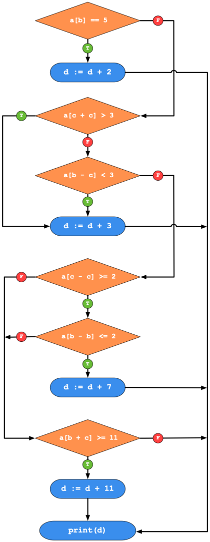
```
a = [2,3,5,7,11]
b = 3
c = 2
result = 0

if a[b] == 5:
    result += 2
elif a[c+c] > 3 or a[b-c] < 3:
    result +=3
elif a[c-c] >= 2 and a[b-b] <= 2:
    result += 7
elif a[b+c] >= 11:
    result += 11
print ('If statement result = ', result)
```

```
If statement result =  3
```

Data structure for the if statement



Flowchart for the if statement



While loop

```
a = 5
b = 3
while a >= b:
    print ("foo")
    a = a + 2
    b = b + 4
    print ("bar")
    a = a + 1
    print ("hello")
print ("world")
```

```
foo
bar
hello
foo
bar
hello
foo
bar
hello
world
```

- Many got this wrong, and there were several different errors
- Some treated $a \geq b$ as if it meant $a > b$
- Some thought the while loop terminates as soon as $a \geq b$ becomes true — the boolean condition is checked at the top of the loop, not continuously as the loop runs

Question: Continue statement

```
i = 1
j = 5
while i < j:
    i = i + 1
    if i == 3:
        continue
    print ('i = ', i)
i =  2
i =  4
i =  5
```

- Answers were all over the place
- That's ok *if you're aware that you don't know what continue does*
- The danger is when you aren't aware
- Continue is dangerous because it's a goto statement that doesn't say explicitly where to go — you need to know how to figure it out
- Continue should be used rarely if at all, just like goto

Question: Break statement

```
i = 1
j = 5
while i < j:
    i = i + 1
    if i == 3:
        break
    print ('i = ', i)

i =  2
```

- Similar to continue: lots of answers, mostly wrong
- Again, that's ok *if you're aware that you don't know what break does*
- Break is dangerous because it's a goto statement that doesn't say explicitly where to go — you need to know how to figure it out
- Break should be used rarely if at all, just like goto

A note about the break statement

- In Python, you can only break out of the innermost loop that contains the break
- If you are in several nested loops, and you want to break out of several of them, *there is no good way to do this in Python*
- Break and continue should be used rarely if at all
- Break and continue are just goto statements, spelled differently
- The disadvantages of goto statements apply to break and continue, only more so
- (Break is commonly used in the C language, because the switch (case) statement in C doesn't work the way you normally want.)

Results

- Some of the primitive operations on lists are widely misunderstood
- There are some misconceptions on what operators $+$ and $+=$ mean when applied to lists
  - Textbooks, web pages, and Stack Overflow also get this wrong quite often
- Some misunderstandings about how nested conditionals work
- The break and continue statements are goto statements where you don't say *where* to go, and this leads to confusion
- Be sure that you know the exact meaning of the language constructs you're using.

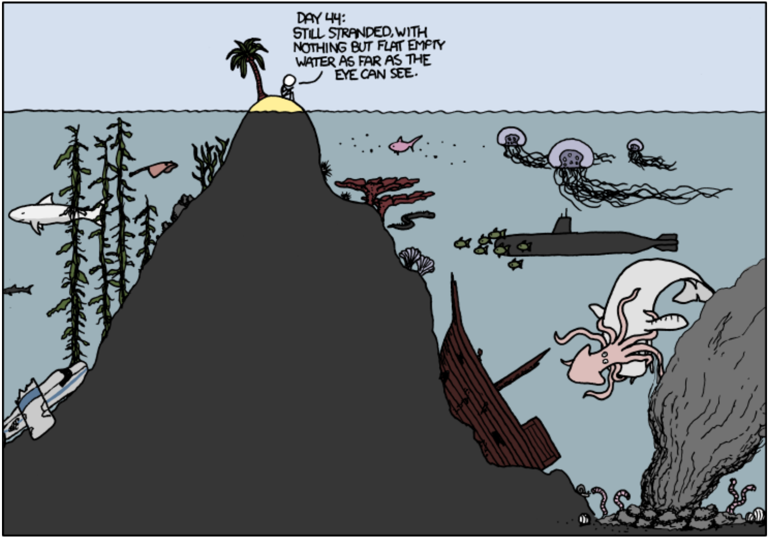
Fundamentals are important

- It's ok if you're not sure about some detail of a programming language
  - *Look it up!*
- But if you have a vague understanding of what a statement really means, it will be hard to write large programs and get them to work reliably
- This can be a source of bugs that are hard to find

Looking forward

- Computer science is a fascinating subject!
- It has connections with just about all other subjects, too
- Learn how to use computers as a tool...
- But keep being curious and keep learning!

Desert island



https://xkcd.com/731/