Computer Systems

Lecture 13

Variables

Dr José Cano Reyes, Dr Mathieu Chollet
School of Computing Science
University of Glasgow
Spring 2024

Outline

- Retrospective

- Variables
  - Static variables
  - Local variables
  - Dynamic variables

- The call stack

- Example: Recursive factorial

Retrospective

- What is a computer program?
- What is a variable?

A review of the ”strange” program

```
    load R1,y[R0]      ; R1 := y
    load R2,x[R0]      ; R2 := x
    add R2,R2,R1
    store R2,x[R0]     ; mem[x] = R2
    lea R3,3[R0]       ; R3 := 3
    lea R4,4[R0]       ; R4 := 4
x   add R5,R3,R3       ; R5 := R3 + R3
    add R0,R0,R7
    trap R0,R0,R0      ; terminate
y   data  0
```

R1: 0 R2: 0 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

```
    load R1,y[R0]        ; R1 := y
    load R2,x[R0]        ; R2 := x
    add R2,R2,R1
    store R2,x[R0]       ; mem[x] = R2
    lea R3,3[R0]         ; R3 := 3
    lea R4,4[R0]         ; R4 := 4
x   add R5,R3,R3         ; R5 := R3 + R3
    add R0,R0,R7
    trap R0,R0,R0        ; terminate
y   data  0
```

What is the value of x?

X is an RRR instruction: its value is the *machine code* of the instruction

-opcode is 0 (add)

-d is 5 (destination R5)

-Ra is 5 (operand1 is R5)

-Rb is 3 (operand2 is R3)

→ mem[x] = $0533

R1: 0 R2: $0533 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

```
      load R1,y[R0]          ; R1 := y
      load R2,x[R0]          ; R2 := x
⟹    add R2,R2,R1
      store R2,x[R0]         ; mem[x] = R2
      lea R3,3[R0]           ; R3 := 3
      lea R4,4[R0]           ; R4 := 4
x     add R5,R3,R3           ; R5 := R3 + R3
      add R0,R0,R7
      trap R0,R0,R0          ; terminate
y     data  0
```

R1: 0　　R2: $0533　　R3: 0　　R4: 0　　R5 : 0

A review of the ”strange” program

```
        load R1,y[R0]         ; R1 := y
        load R2,x[R0]         ; R2 := x
        add R2,R2,R1
  ⟹     store R2,x[R0]        ; mem[x] = R2
        lea R3,3[R0]          ; R3 := 3
        lea R4,4[R0]          ; R4 := 4
x       add R5,R3,R3          ; R5 := R3 + R3
        add R0,R0,R7
        trap R0,R0,R0         ; terminate
y       data  0
```

R1: 0 R2: $0533 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

```
      load R1,y[R0]       ; R1 := y
      load R2,x[R0]       ; R2 := x
      add R2,R2,R1
      store R2,x[R0]      ; mem[x] = R2
      lea R3,3[R0]        ; R3 := 3
      lea R4,4[R0]        ; R4 := 4
x     add R5,R3,R3        ; R5 := R3 + R3
      add R0,R0,R7
      trap R0,R0,R0       ; terminate
y     data  0
```

R1: 0 R2: $0533 R3: 3 R4: 0 R5 : 0

A review of the ”strange” program

```
        load R1,y[R0]        ; R1 := y
        load R2,x[R0]        ; R2 := x
        add R2,R2,R1
        store R2,x[R0]       ; mem[x] = R2
        lea R3,3[R0]         ; R3 := 3
  ⟹    lea R4,4[R0]         ; R4 := 4
x       add R5,R3,R3         ; R5 := R3 + R3
        add R0,R0,R7
        trap R0,R0,R0        ; terminate
y       data 0
```

R1: 0　　R2: $0533　　R3: 3　　R4: 4　　R5 : 0

A review of the ”strange” program

```
   load R1,y[R0]        ; R1 := y
   load R2,x[R0]        ; R2 := x
   add R2,R2,R1
   store R2,x[R0]       ; mem[x] = R2
   lea R3,3[R0]         ; R3 := 3
   lea R4,4[R0]         ; R4 := 4
x  add R5,R3,R3         ; R5 := R3 + R3
   add R0,R0,R7
   trap R0,R0,R0        ; terminate
y  data 0
```

R1: 0 R2: $0533 R3: 3 R4: 4 R5: 6

A review of the "strange" program

```
    load R1,y[R0]        ; R1 := y
    load R2,x[R0]        ; R2 := x
    add R2,R2,R1
    store R2,x[R0]       ; mem[x] = R2
    lea R3,3[R0]         ; R3 := 3
    lea R4,4[R0]         ; R4 := 4
x   add R5,R3,R3         ; R5 := R3 + R3
    add R0,R0,R7
    trap R0,R0,R0        ; terminate
y   data 1
```

What if y=1?

R1: 1 R2: 0 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

```
    load R1,y[R0]       ; R1 := y
    load R2,x[R0]       ; R2 := x
    add R2,R2,R1
    store R2,x[R0]      ; mem[x] = R2
    lea R3,3[R0]        ; R3 := 3
    lea R4,4[R0]        ; R4 := 4
x   add R5,R3,R3        ; R5 := R3 + R3
    add R0,R0,R7
    trap R0,R0,R0       ; terminate
y   data 1
```

What if y=1?

What is the value of x?

X is an RRR instruction: its value is the *machine code* of the instruction

-opcode is 0 (add)

-d is 5 (destination R5)

-Ra is 5 (operand1 is R5)

-Rb is 3 (operand2 is R3)

→ mem[x] = $0533

R1: 1 R2: $0533 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

```
      load R1,y[R0]        ; R1 := y
      load R2,x[R0]        ; R2 := x
      add R2,R2,R1
      store R2,x[R0]       ; mem[x] = R2
      lea R3,3[R0]         ; R3 := 3
      lea R4,4[R0]         ; R4 := 4
x     add R5,R3,R3         ; R5 := R3 + R3
      add R0,R0,R7
      trap R0,R0,R0        ; terminate
y     data 1
```

What if y=1?

R1: 1 R2: $0534 R3: 0 R4: 0 R5 : 0

A review of the "strange" program

```
   load R1,y[R0]      ; R1 := y
   load R2,x[R0]      ; R2 := x
   add R2,R2,R1
   store R2,x[R0]     ; mem[x] = R2
   lea R3,3[R0]       ; R3 := 3
   lea R4,4[R0]       ; R4 := 4
x  add R5,R3,R4       ; R5 := R3 + R4
   add R0,R0,R7
   trap R0,R0,R0      ; terminate
y  data 1
```

What if y=1?

We are modifying the value of x in memory : i.e. we are modifying the program!

What is the instruction with machine code $0534?

-opcode is 0 (add)
-d is 5 (destination R5)
-Ra is 5 (operand1 is R5)
-Rb is 4 (operand2 is R3)
→ add R5,R3,R4

R1: 1 R2: $0534 R3: 0 R4: 0 R5 : 0

A review of the ”strange” program

What if y=1?

```
      load R1,y[R0]        ; R1 := y
      load R2,x[R0]        ; R2 := x
      add R2,R2,R1
      store R2,x[R0]       ; mem[x] = R2
 ⟹   lea R3,3[R0]         ; R3 := 3
      lea R4,4[R0]         ; R4 := 4
x     add R5,R3,R4         ; R5 := R3 + R4
      add R0,R0,R7
      trap R0,R0,R0        ; terminate
y     data 1
```

R1: 1   R2: $0534   R3: 3   R4: 0   R5 : 0

A review of the ”strange” program

What if y=1?

```
     load R1,y[R0]        ; R1 := y
     load R2,x[R0]        ; R2 := x
     add R2,R2,R1
     store R2,x[R0]       ; mem[x] = R2
     lea R3,3[R0]         ; R3 := 3
 ==> lea R4,4[R0]         ; R4 := 4
x    add R5,R3,R4         ; R5 := R3 + R4
     add R0,R0,R7
     trap R0,R0,R0        ; terminate
y    data  1
```

R1: 1   R2: $0534   R3: 3   R4: 4   R5 : 0

A review of the ”strange” program

```
      load R1,y[R0]      ; R1 := y
      load R2,x[R0]      ; R2 := x
      add R2,R2,R1
      store R2,x[R0]     ; mem[x] = R2
      lea R3,3[R0]       ; R3 := 3
      lea R4,4[R0]       ; R4 := 4
x     add R5,R3,R4       ; R5 := R3 + R4
      add R0,R0,R7
      trap R0,R0,R0      ; terminate
y     data 1
```

What if y=1?

This line was modified!

R1: 1 R2: $0534 R3: 3 R4: 4 R5 : 7

A review of the ”strange” program

```
      load R1,y[R0]       ; R1 := y
      load R2,x[R0]       ; R2 := x
      add R2,R2,R1
      store R2,x[R0]      ; mem[x] = R2
      lea R3,3[R0]        ; R3 := 3
      lea R4,4[R0]        ; R4 := 4
x     add R5,R3,R3        ; R5 := R3 + R3
      add R0,R0,R7
      trap R0,R0,R0       ; terminate
y     data  8192
```

What if y=8192?
(=$2000)

R1: $2000   R2: 0   R3: 0   R4: 0   R5 : 0

A review of the ”strange” program

```
     load R1,y[R0]        ; R1 := y
     load R2,x[R0]        ; R2 := x
     add R2,R2,R1
     store R2,x[R0]       ; mem[x] = R2
     lea R3,3[R0]         ; R3 := 3
     lea R4,4[R0]         ; R4 := 4
x    mul R5,R3,R3         ; R5 := R3 * R3
     add R0,R0,R7
     trap R0,R0,R0        ; terminate
y    data   8192
```

What if y=8192?
(=$2000)

$2000 is added to x:
opcode changes from 0 to 2 - add to mul

R1: $2000	R2: $2533	R3: 3	R4: 4	R5 : 9

What is a computer program?

- Beginner's view
  - The computer runs programs, a program is lines of code (Python, C, etc)

- The strange program shows how wrong that view is!
  - The computer doesn’t execute the assembly instructions – it executes machine code from memory

- More sophisticated view
  - The lines of assembly code are input to the assembler which generates the initial value of the machine code
  - When a program is booted, the initial machine code is stored in memory
  - The computer executes the machine language instructions in memory, the original assembly language code (labels and all) no longer exists

- Essential concepts
  - Source code (input to a translator) and object code (output of a translator)
  - Compile time (object code creation) and run time (object code execution)

Outline

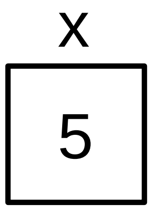
- Retrospective

- Variables
  - Static variables
  - Local variables
  - Dynamic variables

- The call stack

- Example: Recursive factorial

What is a variable?



- Beginner's view
  - A variable is a box with a name that holds a value
  - An expression can use the value in the box, an assignment can modify the value in the box
  - In assembly language, we define a variable with a *data* statement

- More sophisticated view
  - Variables are distinct from variable names: many variables may have the same name
  - A variable has a scope in a program: a region where it corresponds to a particular box
  - Variables do not correspond to *data* statements: they are created and destroyed dynamically as a program runs
  - Initialising a variable is not the same as assigning a value to it

Access to variables

- Different programming languages can manage variables in different ways
  - For example, how variables are accessed in memory

- Three general key aspects
  - The lifetime of a variable: when it is created, when it is destroyed
  - The scope of a variable: which parts of the source program are able to access the variable
  - The location of a variable: what its address in memory is

- The assembler generates the correct object code to access each variable

Three classes of variable

- **Static variables**: Sometimes called global variables
  - Visible through the entire program
- **Local variables**: Sometimes called automatic variables
  - Visible only in a local procedure
- **Dynamic variables**: Sometimes called heap variables
  - Visibility can vary for different variables
  - Used in object oriented and functional languages

Static variables

- The lifetime of a static variable is the entire execution of a program
  - When the program is launched, its static variables are created
  - They exist and retain their values until the program ends

- The scope of a static variable is the entire program
  - Every part of the program (e.g. a procedure) can access them

- The location of static variables is the *global* segment of memory

- Examples
  - In C, you can declare a variable to be static
  - In Pascal, all variables are static/global (if they are not defined locally)
  - So far, in Sigma16 we have used just static variables (defined with *data*)

Combining static variables with code

- The simple way we have been defining variables makes them static

```
    load R1,x[R0]        ; R1 := x
    ...
    trap R0,R0,R0        ; terminate

; Static variables
x data 0
n data 100
```

- There is only one variable x, and one variable n
  - Both exist for the entire program execution

Disadvantages of combining variables and code

- The executable code cannot be shared
  - Suppose two users want to run the program
  - Each needs to have a copy of the entire program, which contains both the instructions and the data
  - This means the instructions are duplicated in memory
  - This is inefficient use of memory

- To avoid the duplication of instructions, we need to separate the data from the instructions

- Modern operating systems organise information into segments
  - A code segment (instructions) is read-only, and can be shared
  - A data segment is read/write, and cannot be shared

Local variables

- The lifetime of a local variable is the function, procedure, method, begin..end block, or a {...} block where are defined
  - They exist while the function, procedure, etc exists
- The scope of a local variable is the function, procedure, etc where is defined
  - A local variable has one name, but there may be many instances of it if the function is recursive
- The location of local variables is a stack frame
  - The variables are accessed using the stack frame register
  - The compiler (or assembler) works out the address of each local variable relative to the address of the stack frame
  - Therefore they cannot be stored in the static data segment

Accessing local variables

- The compiler (or programmer) works out the exact format of the stack frame
- Each local variable has a dedicated location in the stack frame
- Its address (relative to the frame) is used in the load instruction

load R1,7[R14]   ; local variable at position 7, R14 points to stack frame

Dynamic variables

- The lifetime of a dynamic variable does not need to follow the order that stack frames are pushed or popped, it can be irregular
  - A dynamic variable is created explicitly, e.g. using *new* in Java
- The scope of a dynamic variable is not limited to use in just one function
  - They can be several functions or the entire program
- The location of dynamic variables is the heap
  - They can't be kept in the static data segment or the stack
- Used in functional and object-oriented languages

The Heap

- Languages that support dynamic variables (Lisp, Scheme, Haskell, Java) have a region of memory called the heap

- Typically contains a very large number of very small objects

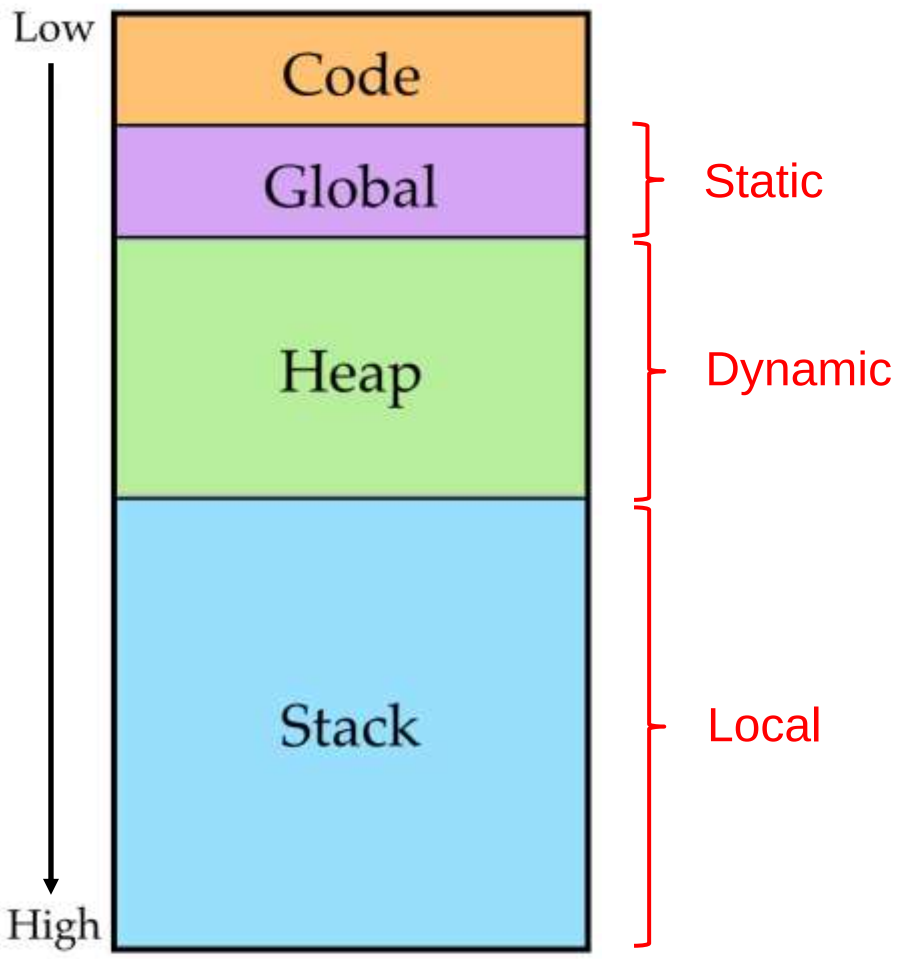
- Contains a free space list, a data structure that points to all the free words of memory

- Maintained by the language “runtime system”, not by the operating system

- When you do a *new*, a (small) amount of memory is allocated from the heap and a pointer (address) to the object is returned

- When the object is no longer required, the memory used to hold it is linked back into the free space list

Memory



Outline

- Retrospective
- Variables
  - Static variables
  - Local variables
  - Dynamic variables
- The call stack
- Example: Recursive factorial

The call stack

- Each procedure call pushes information on the stack, a stack frame
- When a procedure returns, its stack frame is popped (removed) from the stack
- The same register is used as the stack pointer (address of the top stack frame)
  - Each architecture uses a standard register to keep the stack pointer
  - In Sigma16, R14 is the stack pointer
  - When you call, you push a new stack frame and increase R14
  - As a procedure runs, it access its data via R14
  - When you return, you set R14 to the stack frame below
- Remember that the stack frame contains
  - A pointer to the previous stack frame, the return address (the value of R13)
  - The saved registers, local variables

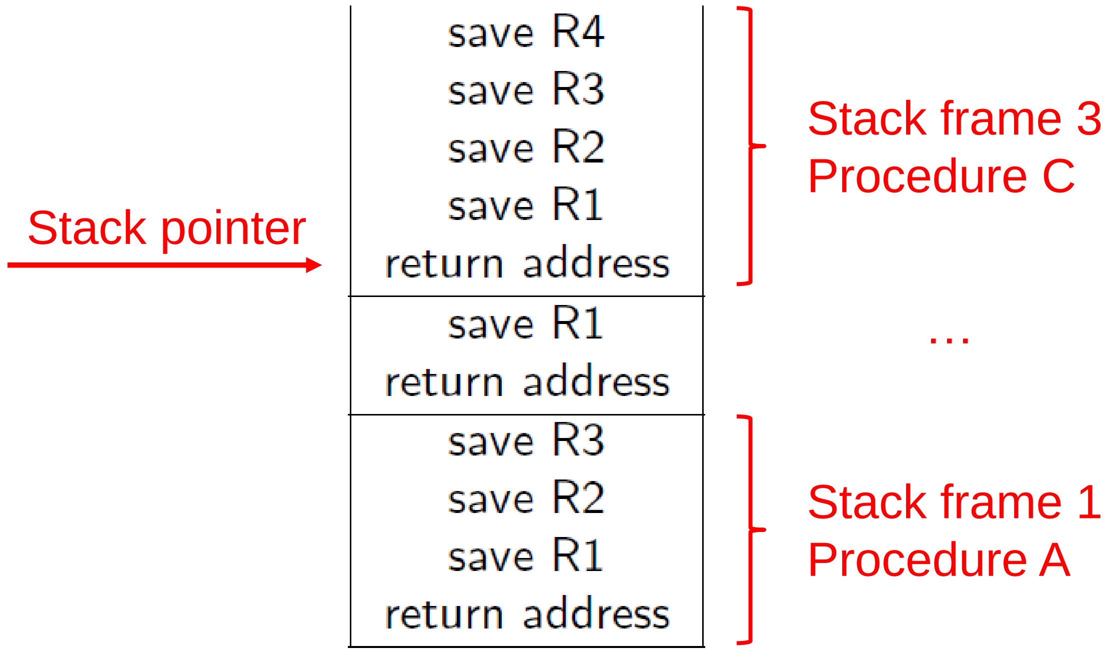
Simplest stack: return addresses

- Just save the return address on the stack

return address
return address
return address
return address

Saved registers

- Save the registers the procedure needs to use on the stack, and restore them before returning
  – This way the procedure won't crash the caller



Dynamic links

- **Problem**: since each stack frame can have a different size, how do we pop the top one o the stack?

- **Simplest solution**: each stack frame contains a pointer (called dynamic link) to the one below

4	save R3
3	save R2
2	save R1
1	return address
0	dynamic link
2	save R1
1	return address
0	dynamic link
3	save R2
2	save R1
1	return address
0	dynamic link
...	

Local variables

- The procedure keeps its local variables on the stack (no data statements)

6	y
5	x
4	save R3
3	save R2
2	save R1
1	return address
0	dynamic link
3	pqrs
2	save R1
1	return address
0	dynamic link
5	b
4	a
3	save R2
2	save R1
1	return address
0	dynamic link
...	

Static links for scoped variables

- Some programming languages require additional information, “static link"

7	y
6	x
5	save R3
4	save R2
3	save R1
2	static link
1	return address
0	dynamic link
4	pqrs
3	save R1
2	static link
1	return address
0	dynamic link
6	b
5	a
4	save R2
3	save R1
2	static link
1	return address
0	dynamic link

Accessing a word in the stack frame

- We need a "map" showing the format of a stack frame
  - It is described in comments (similar to the register usage comments)

- Suppose that local variable "x" is kept at position 7 in the stack frame, so to access the variable

load R1,7[R14]   ; R1 := x

store R1,7[R14]   ; R1 := x

- They are called local variables because every call to a procedure has its own private copy

Example from factorial program

- Memory map of the stack frame for the factorial function
- The comments document the structure of a stack frame for the program

```
; Structure of stack frame for factorial function
;       6[R14]    origin of next frame
;       5[R14]    save R4
;       4[R14]    save R3
;       3[R14]    save R2
;       2[R14]    save R1 (parameter n)
;       1[R14]    return address
;       0[R14]    pointer to previous stack frame
```

Outline

- Retrospective
- Variables
  - Static variables
  - Local variables
  - Dynamic variables
- The call stack
- Example: Recursive factorial

Recursive factorial

- In the Sigma16 examples, there is a program called factorial
- This program illustrates the full stack frame technique
- It uses recursion, i.e. a function that calls itself
- **Note**: the best way to compute a factorial is with a simple loop, not with recursion
- But recursion is an important technique, and it's better to study it with a simple example (like factorial) rather than a complicated “real world” example

About the factorial program

- Comments are used to identify the program, describe the algorithm, and document the data structures
- Blank lines and full-line comments organise the program into small sections
- The caller just uses jal to call the function
- The function is responsible for building the stack frame, saving and restoring registers
- The technique of using the stack for functions is general, and can be used for large scale programs

Statement of problem, and register usage

```
;---------------------------------------------------------------------
; Program Factorial
;---------------------------------------------------------------------

; This program for the Sigma16 architecture uses a recursive function
; to compute x! (factorial of x), where x is defined as a static
; variable.

; The algorithm uses a recursive definition of factorial:
;   if n <= 1
;       then factorial n = 1
;       else factorial n = n * factorial (n-1)

; Register usage
;   R15 is reserved by architecture for special instructions
;   R14 is stack pointer
;   R13 is return address
;   R2, R3, R4 are temporaries used by factorial function
;   R1 is function parameter and result
;   R0 is reserved by architecture for constant 0
```

Format of main program stack frame

```
;----------------------------------------------------------------------
; Main program

; The main program computes result := factorial x and terminates.

; Structure of stack frame for main program
;    1[R14]   origin of next frame
;    0[R14]   pointer to previous stack frame = nil
```

Main program initialisation

```
; Initialise stack
        lea     R14,stack[R0]        ; initialise stack pointer
        store   R0,0[R14]            ; previous frame pointer := nil
```

Main program calls factorial

```
; Call the function to compute factorial x
        load    R1,x[R0]            ; function parameter := x
        store   R14,1[R14]          ; point to current frame
        lea     R14,1[R14]          ; push stack frame
        jal     R13,factorial[R0]   ; R1 := factorial x
```

Main program finishes

```
; Save result and terminate
        store   R1,result[R0]       ; result := factorial x
        trap    R0,R0,R0            ; terminate
```

Description of factorial function

```
;-------------------------------------------------------------
factorial
; Function that computes n!
;   Input parameter n is passed in R1
;   Result is returned in R1
```

Format of stack frame for factorial

```
; Structure of stack frame for fact function
;     6[R14]   origin of next frame
;     5[R14]   save R4
;     4[R14]   save R3
;     3[R14]   save R2
;     2[R14]   save R1 (parameter n)
;     1[R14]   return address
;     0[R14]   pointer to previous stack frame
```

Factorial: build stack frame

```
; Create stack frame
        store   R13,1[R14]            ; save return address
        store   R1,2[R14]             ; save R1
        store   R2,3[R14]             ; save R2
        store   R3,4[R14]             ; save R3
        store   R4,5[R14]             ; save R4
```

Factorial: check for base or recursion case

```
; Initialise
     lea      R2,1[R0]                ; R2 := 1

; Determine whether we have base case or recursion case
     cmp   R1,R2                      ; compare n, 1
     jumpgt recursion[R0]             ; if n>1 then go to recursion
```

Factorial: base case

```
; Base case.  n<=1 so the result is 1
       lea     R1,1[R0]              ; factorial n = 1
       jump    return[R0]            ; go to end of function
```

Factorial: recursion case

```
; Recursion case.  n>1 so factorial n = n * factorial (n-1)
recursion
      sub    R1,R1,R2              ; function paramemter := n-1

; Call function to compute factorial (n-1)
      store  R14,6[R14]            ; point to current frame
      lea    R14,6[R14]            ; push stack frame
      jal    R13,factorial[R0]     ; R1 := factorial (n-1)
      load   R2,2[R14]             ; R2 := saved R1 = n
      mul    R1,R2,R1              ; R1 := n * fact (n-1)
```

Factorial: restore registers and return

```
; Restore registers and return; R1 contains result
return
        load    R2,3[R14]           ; restore R2
        load    R3,4[R14]           ; restore R3
        load    R4,5[R14]           ; restore R4
        load    R13,1[R14]          ; restore return address
        load    R14,0[R14]          ; pop stack frame
        jump    0[R13]              ; return
```

Static data area

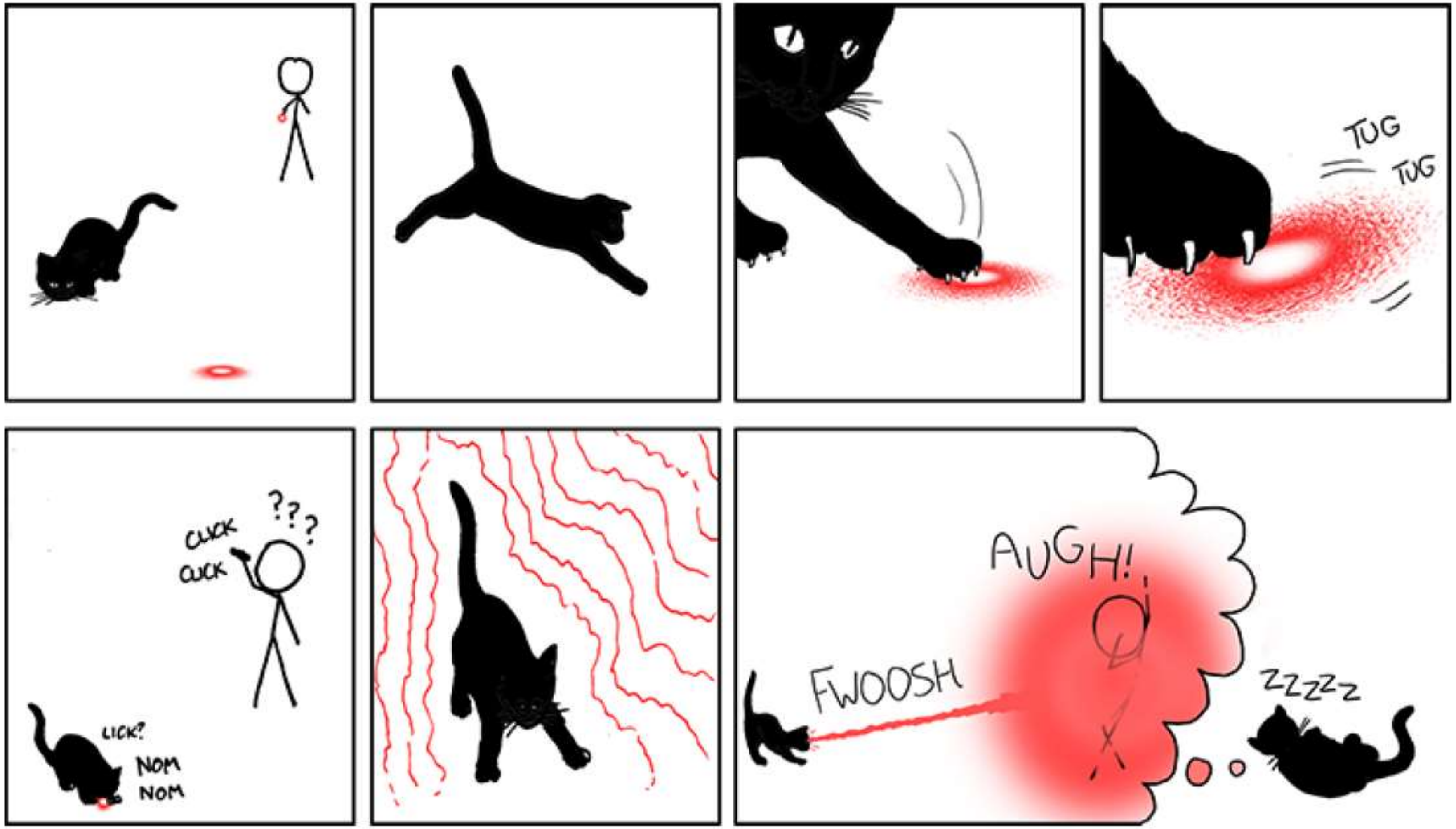
```
;------------------------------------------------------------------
; Static data segment

x         data     5    ; input x
result    data     0    ; x! (x factorial)
stack     data     0    ; stack extends from here on...
```

Summary

- Variables defined with *data* statement are static
  - Each static variable must have a unique name
  - Static variables exist through entire execution of program

- Variables defined in a *procedure* are local
  - Different procedures can use the same name for different variables
  - Local variables are kept in the stack frame (accessed using R14)
  - Call (push stack frame), return (pop stack frame)
  - R14 points to current stack frame

Laser pointer



https://xkcd.com/729/