COMPILERS
Activation Records

hussein suleman

uct csc3003s 2008

Subprogram Invocation Mechanics

- Save status of caller.
- Process parameters.
- Save return address.
- Jump to called subprogram.
- … do stuff …
- Process value-result/result parameters and function return value(s).
- Restore status of caller.
- Jump back to caller's saved position.

Frames / Activation Records

□ An activation record is the layout of data needed to support a call to a subprogram.

□ For languages that do not allow recursion, each subprogram has a single fixed activation record instance stored in memory (and no links).
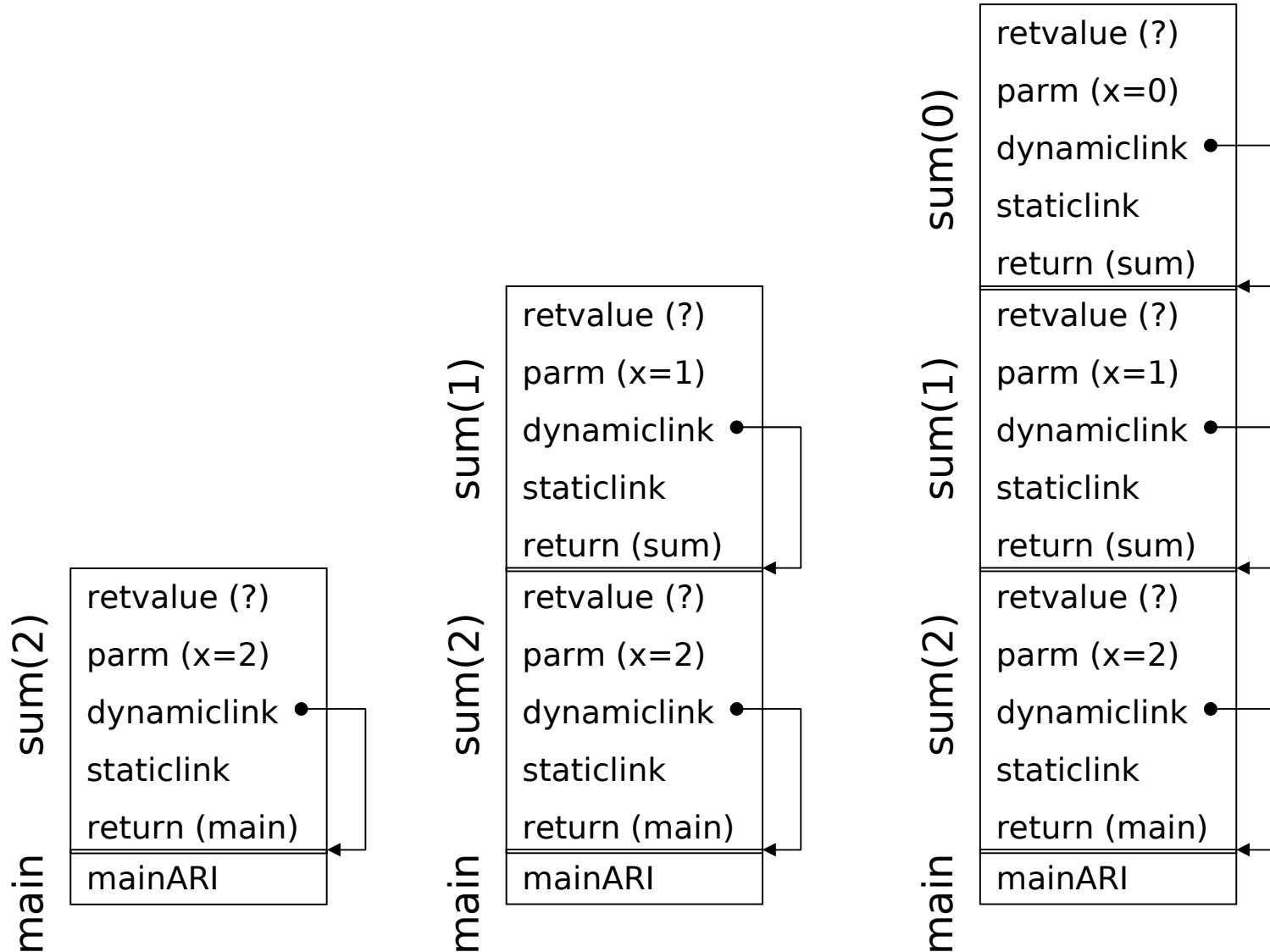
Function return value
Local variables
Parameters
Dynamic link
Static link
Return address

Stack-based Recursion

- When recursion is implemented using a stack, activation records are pushed onto the stack at invocation and popped upon return.

- Example:

```
int sum ( int x )
{
    if (x==0) return 0;
    else return (x + sum (x-1));
}
void main ()
{ sum (2); }
```

Recursion Activation Records
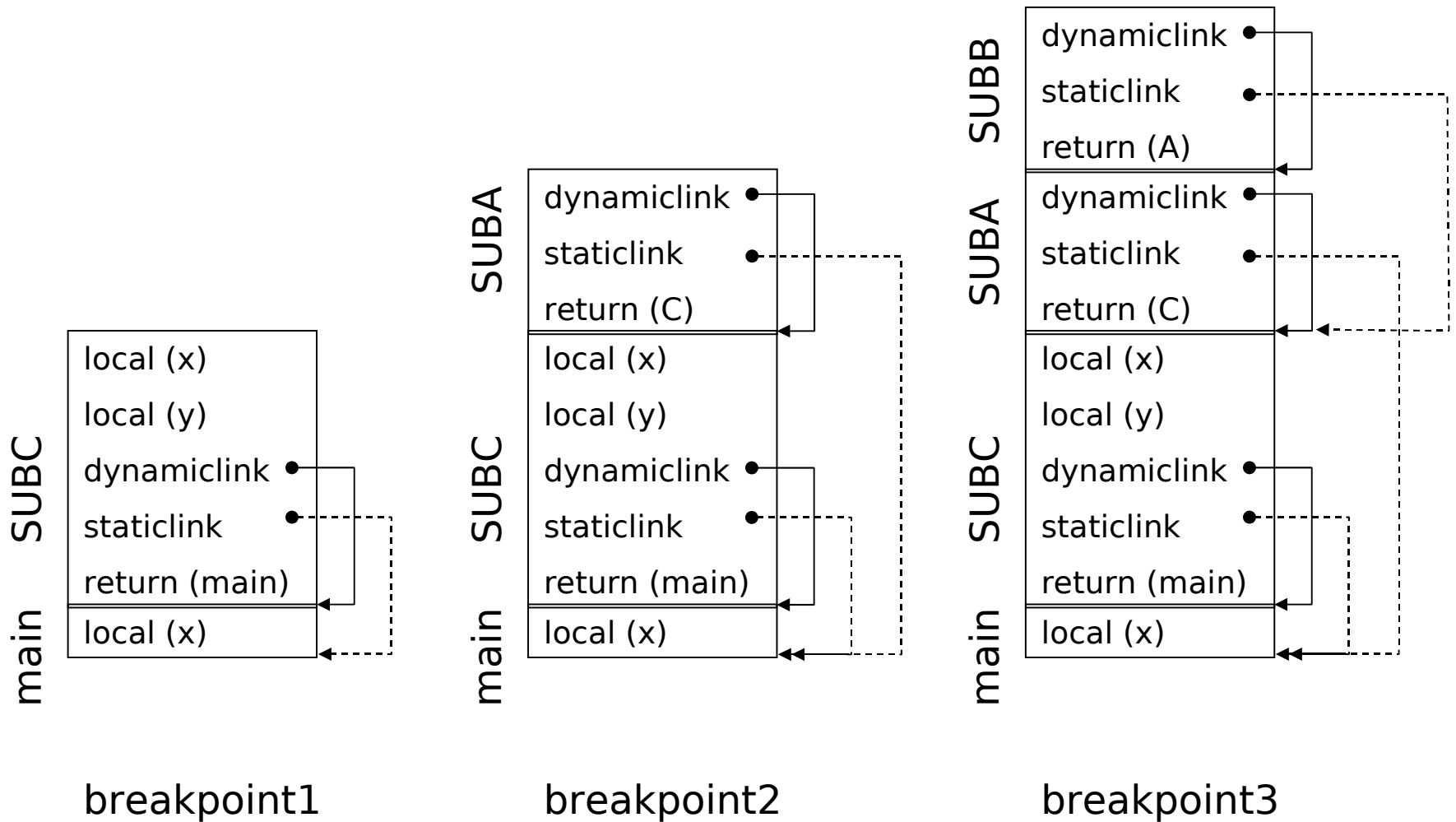
Non-local References

- To access non-local names in statically-scoped languages, a program must keep track of the current referencing environment.

- Static chains
  - Link a subprogram's activation record to its static parent.

- Displays
  - Keep a list of active activation records.

Non-local Reference Example

- □ Example:
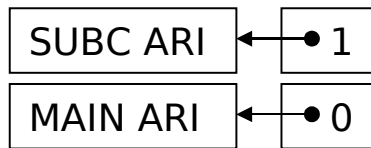
```
main {
    int x;
    sub SUBA {
        sub SUBB {
            x = 1;          ⟵  breakpoint3
        }
        SUBB;               ⟵  breakpoint2
    }
    sub SUBC {
        int x;
        int y;
        SUBA;               ⟵  breakpoint1
    }
    SUBC;                   ⟵  breakpoint0
}
```
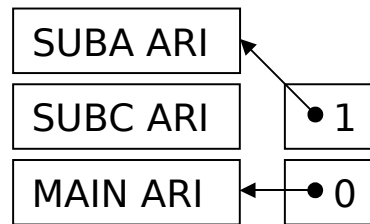
Static Chains

SUBC / **main** (breakpoint1)

local (x)
local (y)
dynamiclink ●
staticlink ●
return (main)
local (x)

SUBA / **SUBC** / **main** (breakpoint2)

dynamiclink ●
staticlink ●
return (C)
local (x)
local (y)
dynamiclink ●
staticlink ●
return (main)
local (x)

SUBB / **SUBA** / **SUBC** / **main** (breakpoint3)

dynamiclink ●
staticlink ●
return (A)
dynamiclink ●
staticlink ●
return (C)
local (x)
local (y)
dynamiclink ●
staticlink ●
return (main)
local (x)

breakpoint1   breakpoint2   breakpoint3

Displays

| SUBC ARI | ← | ● 1 |
| MAIN ARI | ← | ● 0 |

stack   display

breakpoint1

SUBA ARI		
SUBC ARI	↖	● 1
MAIN ARI	←	● 0

stack   display

breakpoint2

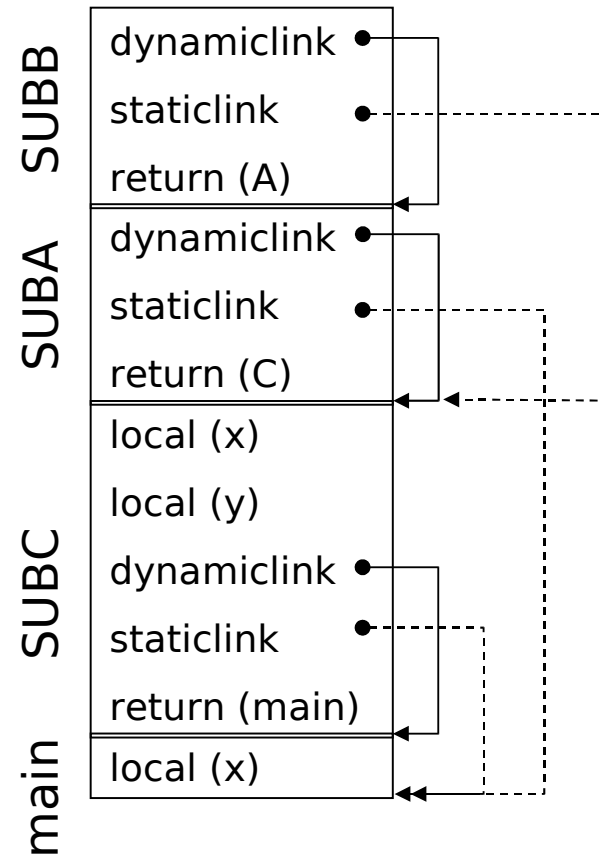| SUBB ARI | | |
SUBA ARI	↖	● 2
SUBC ARI		● 1
MAIN ARI	←	● 0

stack   display

breakpoint3

Static Chains vs. Displays

- Static chains require more indirect addressing – displays require a fixed amount of work.

- Displays require pointer maintenance on return – static chains do not.

- Displays require "backing up" of display pointer – static chains require static links in each activation record.

Dynamic Scoping

- Dynamically scoped languages can be implemented using:
- Deep Access
  - Follow the dynamic chains to find most recent non-local name definition.
- Shallow Access
 - Maintain a separate stack for each name.

Deep Access

- At breakpoint3, by following dynamic links from SUBB, the closest definition of x is in SUBC.

- (Remember that for static scoping, by following static links, the closest definition is in main.)

SUBB	dynamiclink
	staticlink
	return (A)
SUBA	dynamiclink
	staticlink
	return (C)
SUBC	local (x)
	local (y)
	dynamiclink
	staticlink
	return (main)
main	local (x)

breakpoint3

Frame Pointers

☐ Stack frames are usually supported by:
- stack pointer - points to top of stack
- frame pointer - points to top of previous frame

frame
pointer

AR f

AR g

stack
pointer

After call to h()

AR f

AR g

frame
pointer

AR h
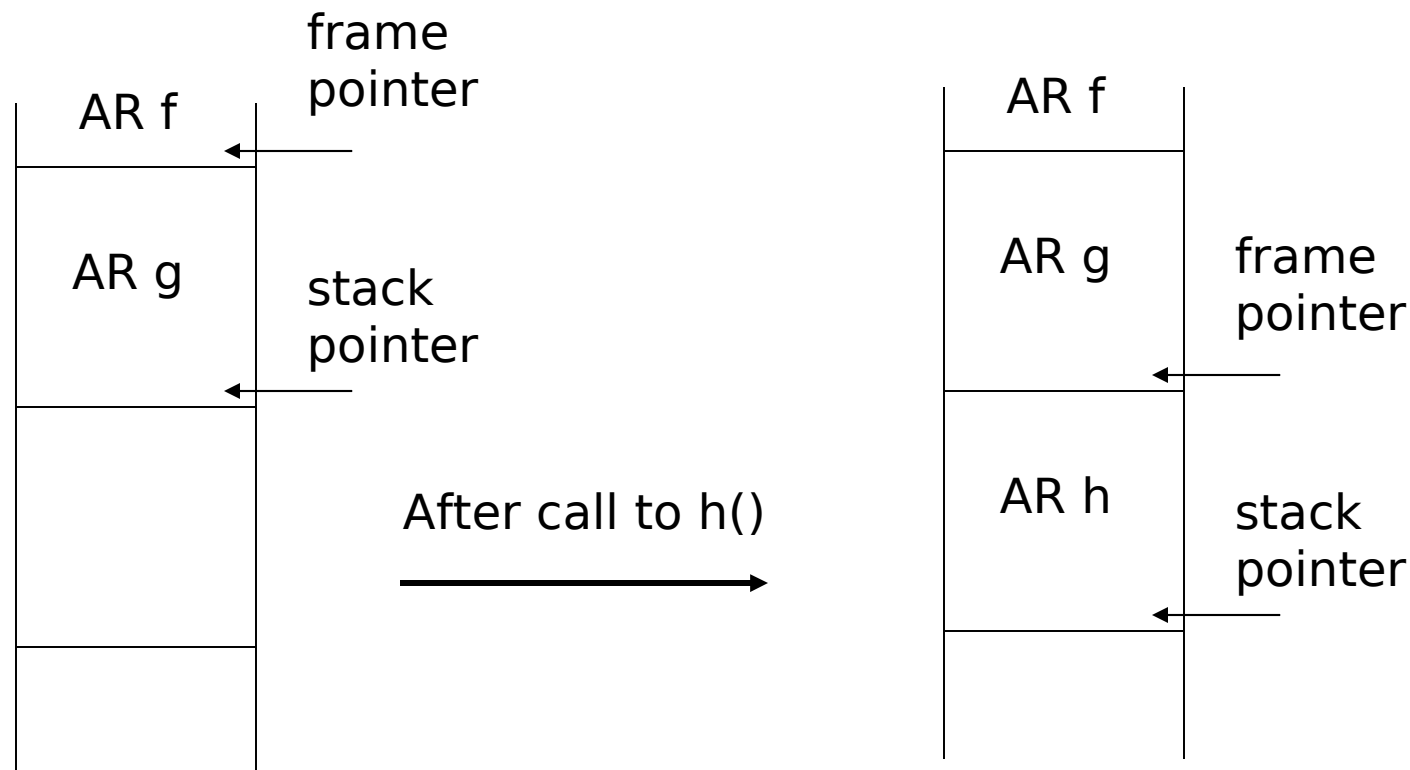
stack
pointer

View Shifts

- On a Pentium machine,
    - M[SP + 0] <-- FP
        - save old frame pointer
    - FP <-- SP
        - move frame pointer to top of stack
    - SP <-- SP - K
        - move stack pointer to end of new frame

- On machines which use registers for frame optimisation, remember to save registers in temporary variables.

Register Handling

- One set of registers are typically used by many subprograms, so a value expected by one may be overwritten by another.
- Solution:
  - Make it the responsibility of the caller to save registers first (caller-save)
  - Make it the responsibility of the callee to save registers first (callee-save)
- Optimise which registers need to be saved as some values can be thrown away.

Parameter Passing

- Registers are more efficient than copying every parameter to the stack frame.
  - Registers are limited so pass first k parameters in registers and rest in frame.
- Nested subprogram calls require saving and restoring so there is dubious cost savings!
  - leaf procedures, different registers, done with variables, register windows
- How does C support varargs ?

Return Addresses

□ Traditionally a stack frame entry.

□ More efficient to simply use a register.

■ Same saving procedure necessary as before for non-leaf subprograms.

Temporaries and Labels

- Each time a local variable is encountered, a unique *temporary* name is generated – this temporary will eventually map to either a register or a memory location (usually on the stack).

- Each time a subprogram is encountered, a unique *label* is generated.

- These must be unique to prevent naming conflicts - the optimiser will deal with efficiency.

Frame Implementation 1/2

- A Frame class corresponds to the frame for each subprogram.
    - During translation, frames are created to track variables and generate prologue/epilogue code.
- Frame can be an abstract class with instantiations for different machine architectures.
    - Each instantiation must know how to implement a "view shift" from one frame to another.

Frame Implementation 2/2

- Each time a local variable is defined, a method of Frame can be called to allocate space appropriately (on stack frame or in registers).
  - f.allocLocal (false)
  - Parameter indicates if variable requires memory (escapes) or not - should we allocate stack space or temporary?
- Allocating a temporary for each variable can be slow - future stages will optimise by reusing both registers and space.

Stack vs. Registers

- Why use registers?
  - Faster and smaller code
- If registers are so great, why use stack?
  - variables used/passed by reference
  - nested subprograms
  - variable is not simple or just too big
  - arrays
  - registers are needed for other purposes
  - too many variables