# CMPT 379
# Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop



## Run-time Support

- Tracking variable usage is done using activation or liveness analysis
- Functions or procedures have more complex activation behaviour
- Problem: functions can be recursive
- This means each function activation has to keep it’s locals and parameters distinct

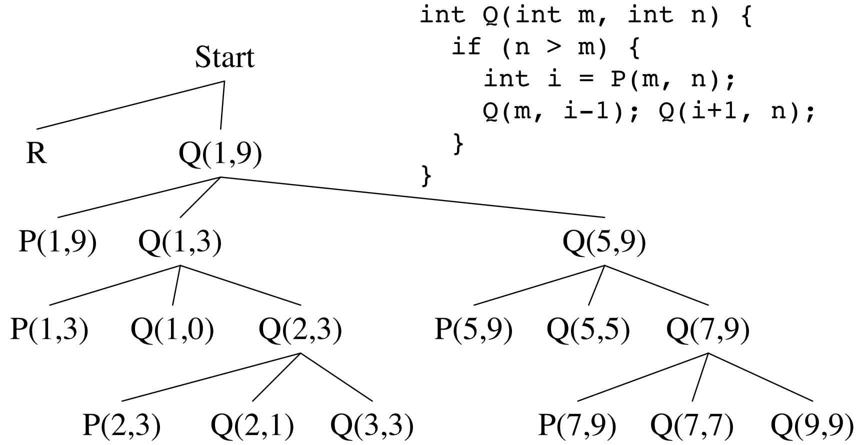
# Activation Trees

- An activation of a function is a particular invocation of that function
- Each activation will have particular values for the function parameters
- Each activation can call another activation before it becomes inactive
- The sequence of function calls can be represented as an *activation tree*



# Activation Tree

```
int Q(int m, int n) {
  if (n > m) {
    int i = P(m, n);
    Q(m, i-1); Q(i+1, n);
  }
}
```



# Problems with Functions

- Recursive functions
- If a function has local variables, and if it calls another function: what happens to locals after control returns
- Function can access non-local (global) variables
- Parameter passing into a function



# More problems

- Can we pass functions as parameters?
- Can functions be returned as the result of a function?
- Storage allocation within a function
- Is de-allocation to be done by the programmer before leaving the function
- Dangling pointers

# Activation Records

- Information for a single execution of a function is called an *activation record* or *procedure call frame*
- A frame contains:
  - Temporary local register values for caller
  - Local data
  - Snapshot of machine state (important registers)
  - Return address
  - Link to global data
  - Parameters passed to function
  - Return value for the caller



# Storage Allocation for Functions

- Static Allocation
  - Layout all storage for all data objects at compile time
  - Essentially every variable is stored globally
  - But the symbol table can still control local activation and de-activation of variables
  - Very restricted recursion is allowed
  - Fortran 77

# Storage Allocation for Functions

- Stack Allocation √
  - Storage for recursive functions is organized as a stack: last-in first-out (LIFO) order
  - Activation records are associated with each function activation
  - Activation records are pushed onto the stack when a call is made to the function
  - Size of activation records can be fixed or variable



# Storage Allocation for Functions

- Stack Allocation √
  - Sometimes a minimum size is required
  - Variable length data is handled using pointers
  - Locals are deleted after activation ends
  - Caller locals are reinstated and execution continues
  - C, Pascal and most modern programming languages

## Storage Allocation for Functions

- Heap Allocation
  - In some special cases stack allocation is not possible
  - If local variables must be retained after the activation ends
  - If called activation outlives the caller
  - Anything that violates the last-in first-out nature of stack allocation e.g. closures in Lisp and other functional PLs



## Heap Allocation

```
class Ret {
    int a; a = 10;
    fun foo (int m) {
        int addm (int n) { return (a+m+n); }
        return addm;
    }
    int main() {
        callout("print_int", (foo(2))(3));
    }
}
```

# Storage Allocation for Functions

- Function Composition: (f•g)(x) = f(g(x))

```
class Compose {
    fun sq (int x) { return (x * x); }
    fun f (fun m) { return (m•h); }
    fun h () { return sq; }
    fun g (fun z) { return (sq•z); }
    int main() {
        fun v = g•h;
        callout("print_int", (v())(3));
    }
}
```



# Storage Allocation for Functions

- Function Composition: (f•g)(x) = f(g(x))

```
class Compose {
    fun sq (int x) { return (x * x); }
    fun f (fun m) { return (m•h); }
    fun h () { return sq; }
    fun g (fun z) { return (sq•z); }
    int main() {
        fun v = g•h;
        callout("print_int", (v())(3));
    }
}
```

v = g•h

v() = (g•h)()

v() = g(h())

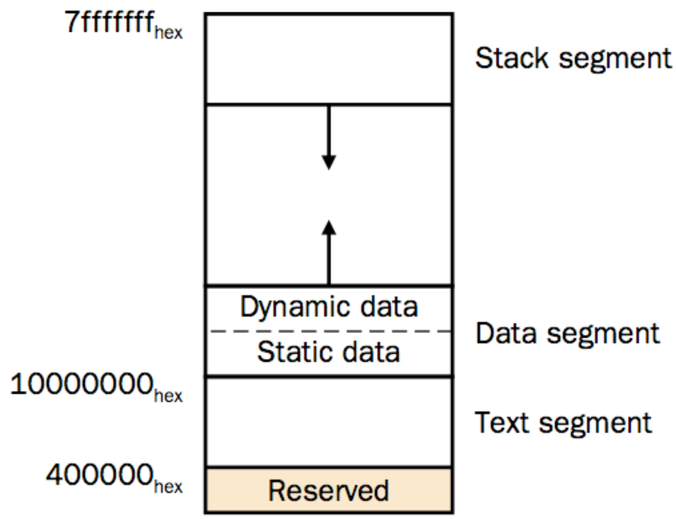
v() = g(sq)

v() = (sq•sq)

v()(3) = (sq•sq)(3)

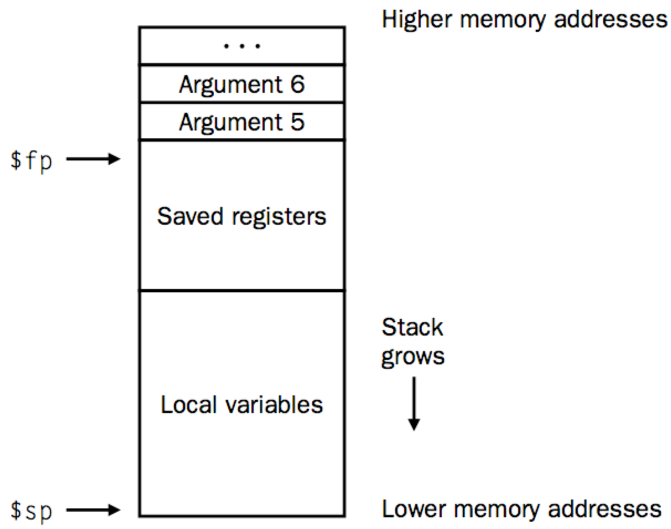
v()(3) = (sq(sq(3))

# Run-time Memory



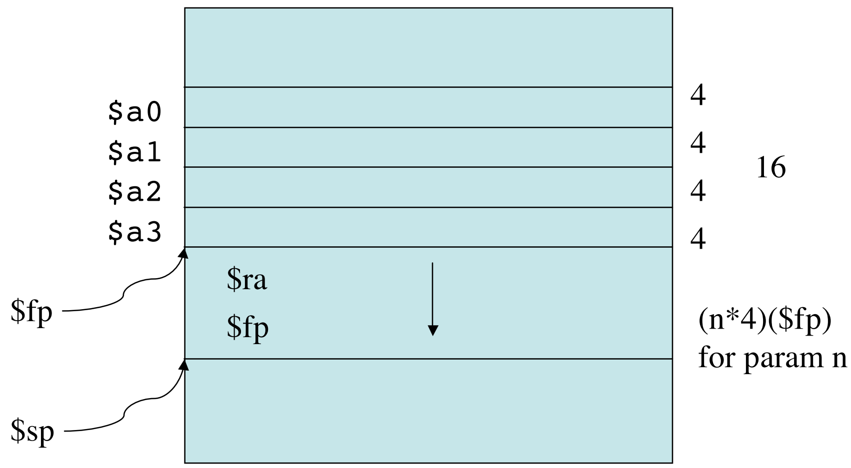

11/5/0 15

# Stack frame

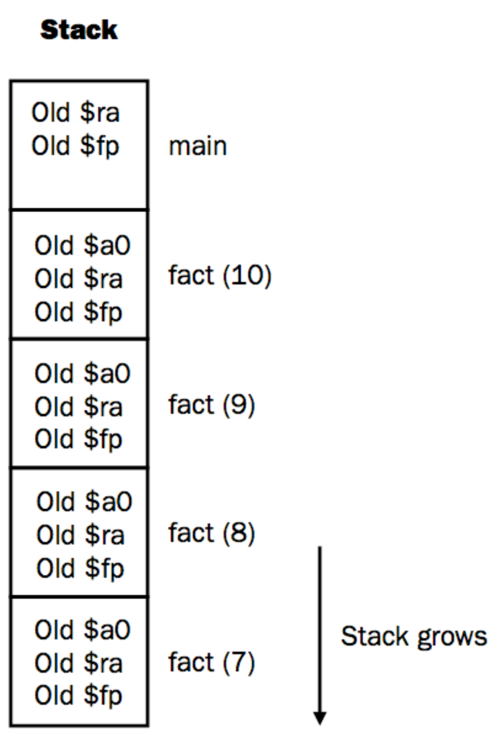



11/5/07 16

# Example: MIPS stack frame




11/5/07 17




11/5/07 18

# Parameter Passing Conventions

- Differences based on:
  - The parameter represents an r-value (the rhs of an expr)
  - An l-value
  - Or the text of the parameter itself
- Call by Value
  - Each parameter is evaluated
  - Pass the r-value to the function
  - No side-effect on the parameter



# Parameter Passing Conventions

- Call by Reference
  - Also called call by address/location
  - If the parameter is a name or expr that is an l-value then pass the l-value
  - Else create a new temporary l-value and pass that
  - Typical example: passing array elements a[i]

# Parameter Passing Conventions

- Copy Restore Linkage
  - Pass only r-values to the called function (but keep the l-value around for those parameters that have it)
  - When control returns back, take the r-values and copy it into the l-values for the parameters that have it
  - Fortran
- Call by Name
  - Function is treated like a macro (a #define) or in-line expansion
  - The parameters are literally re-written as passed arguments (keep caller variables distinct by renaming)



# Parameter Passing Conventions

- Lazy evaluation
  - In some languages, call-by-name is accomplished by sending a function (also called a thunk) instead of an r-value
  - When the r-value is needed the function is called with zero arguments to produce the r-value
  - This avoids the time-consuming evaluation of r-values which may or may not be used by the called function (especially when you consider short-circuit evaluation)
  - Used in lazy functional languages

# Parameter Passing Conventions

- Call-by-need
  - Similar to lazy evaluation, but more efficient
  - To avoid executing similar r-values multiple times, some languages used a memo slot to avoid repeated function evaluations
  - A function parameter is only evaluated when used inside the called function
  - When used multiple times there is no overhead due to the memo table
  - Haskell



# Summary

- Run-time support for functions
- Dealing with (potentially infinite) recursion
- Activation records for each function invocation
- Storage allocation for activation records in recursive function calls
- Stack allocation is easiest to implement while retaining recursion
- Functional PLs use heap allocation