The General Semantics of Calls and Returns

- The subprogram call and return operations of a language are together called its *subprogram linkage*
- A subprogram call has numerous actions associated with it
  - Parameter passing methods
  - Static local variables
  - Execution status of calling program
  - Transfer of control
  - Subprogram nesting

Implementing “Simple” Subprograms:

- Subprograms cannot be nested
- All local variables are static
- e.g. Subprograms in early versions of Fortran

Implementing "Simple" Subprograms: Call Semantics

- Save the execution status of the caller
- Carry out the parameter-passing process
- Pass the return address to the callee
- Transfer control to the callee

Implementing "Simple" Subprograms: Return Semantics

- If pass-by-value-result parameters are used, move the current values of those parameters to their corresponding actual parameters
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

Storage required for call/return actions

- Status information about the caller
- Parameters
- Return address
- Functional value for function subprograms

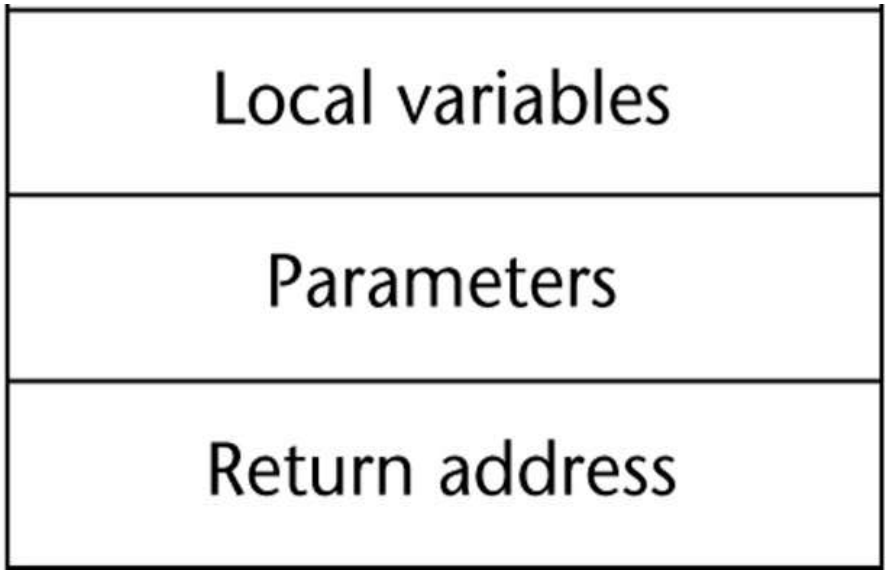
- These along with local variables and the subprogram code, form the complete collection of information a subprogram needs to execute and then return to the caller

Implementing "Simple" Subprograms: Parts

- Two separate parts: the actual code and the noncode part (local variables and data that can change)
- The format, or layout, of the noncode part of an executing subprogram is called an *activation record*
- An *activation record instance (ARI)* is a concrete example of an activation record (the collection of data for a particular subprogram activation)

An Activation Record for "Simple" Subprograms

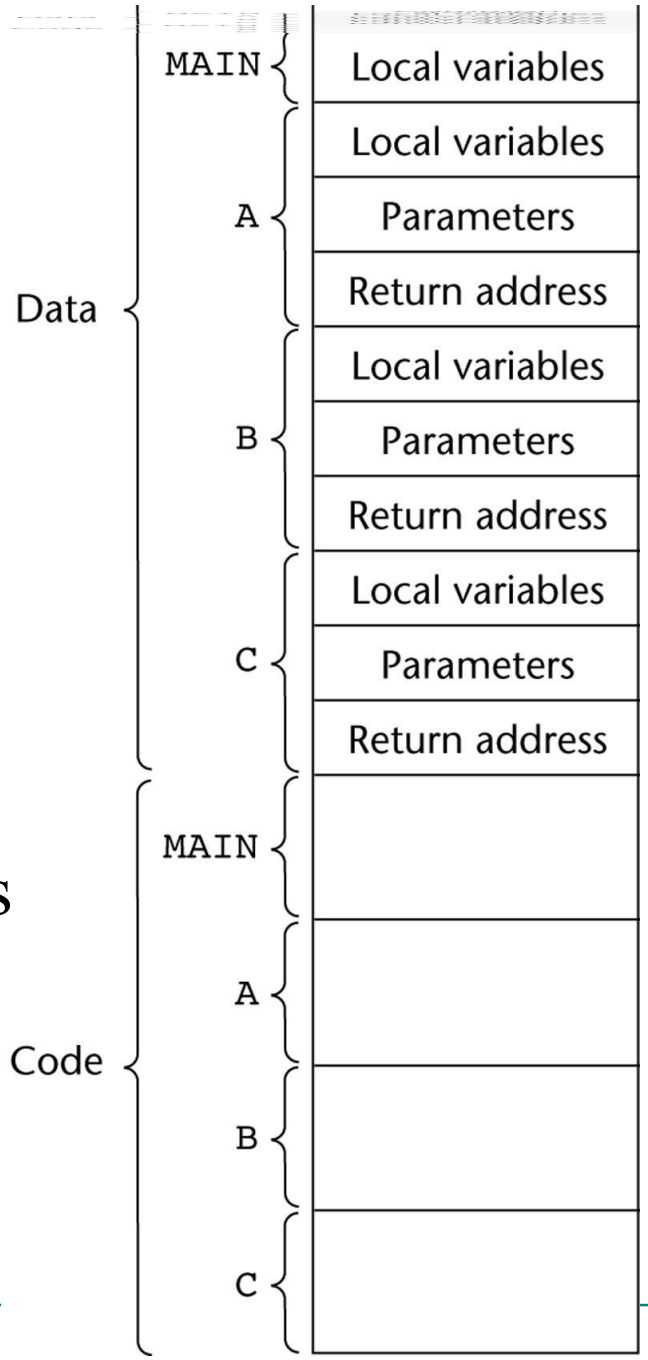
Because languages with simple subprograms do not support recursion, there can be only one active version of a given subprogram at a time
Therefore, there can be only a single instance of the activation record for a subprogram



Since activation record instance of a simple subprogram has fixed size it can be statically allocated
It could be also attached to the code part

Note: In the rest of the slides the saved execution status of the caller will be omitted

Code and Activation Records of a Program with "Simple" Subprograms

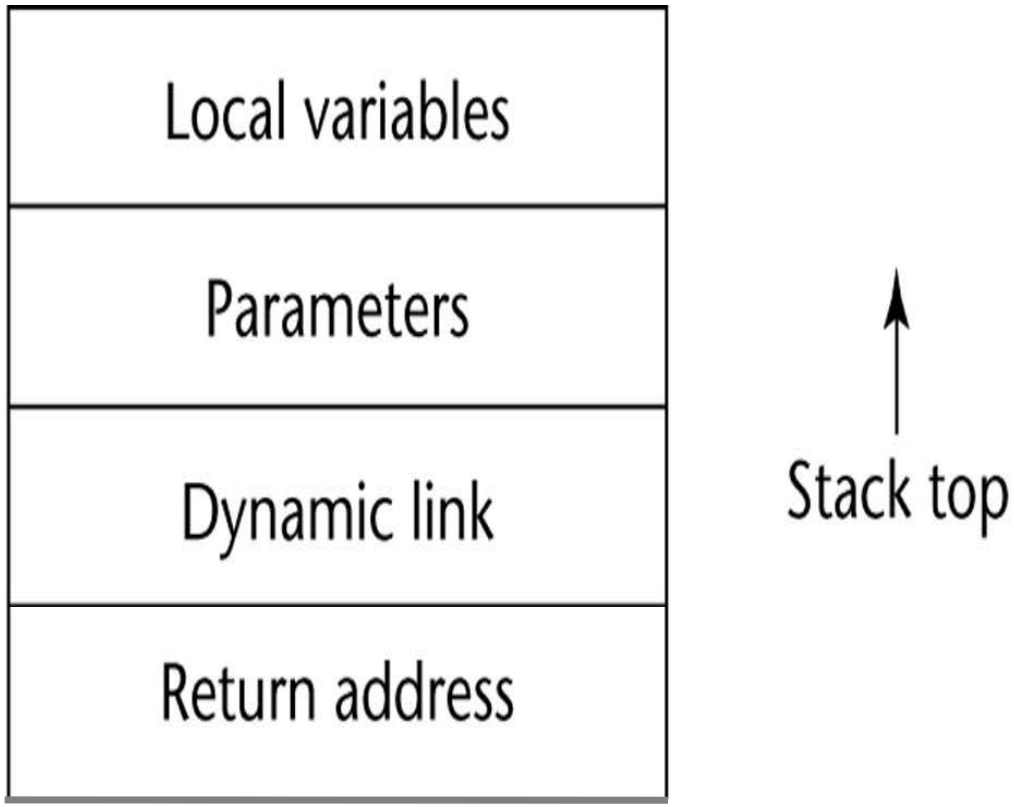


Note that code could be attached to ARIs
Also, the four program units could be compiled at different times
Linker put the compiled parts together when it is called for the main program

Implementing Subprograms with Stack-Dynamic Local Variables

- More complex activation record
  - The compiler must generate code to cause implicit allocation and de-allocation of local variables
  - Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

Typical Activation Record for a Language with Stack-Dynamic Local Variables



Return address: pointer to the code segment of the caller and an offset address in that code segment of the instruction following the call

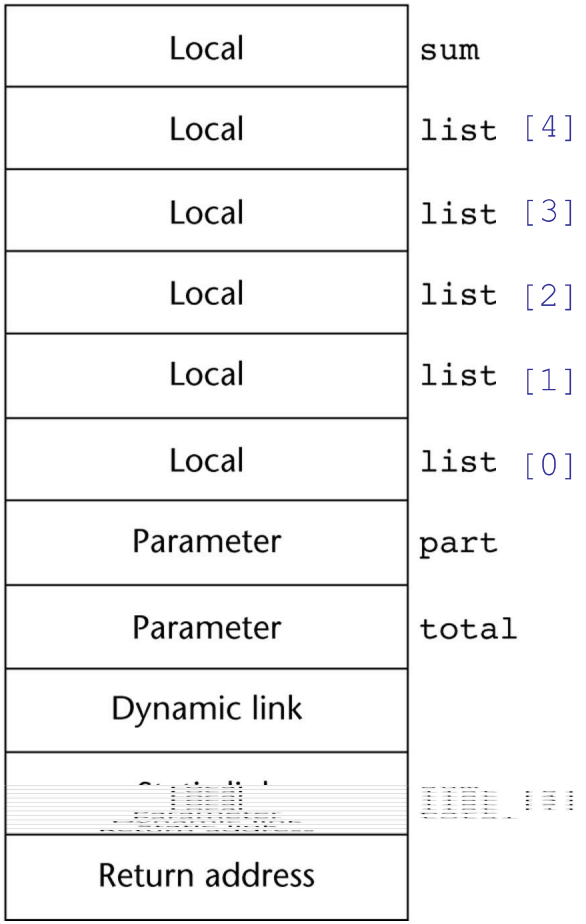
Dynamic link: pointer to the top of the activation record instance of the caller
In static scoped languages this link is used in destruction
The stack top set to the value of old dynamic link

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic
- An activation record instance is dynamically created when a subprogram is called
- Run-time stack

An Example: C Function

```
void sub(float total, int part)
{
   int list[4];
   float sum;
   …
}
```



An Example Without Recursion

```
void A(int x) {
    int y;
    ...
    C(y);
    ...
}
void B(float r) {
    int s, t;
    ...
    A(s);
    ...
}
void C(int q) {
    ...
}
void main() {
    float p;
    ...
    B(p);
    ...
}
```

main calls B
B calls A
A calls C

An Example Without Recursion