Subprogram Implementation

COMP3220 – Principle of Programming Languages

Zhitao Gong

2016 Spring

Outline

Linkage

Subprogram Linkage The subprogram call and return operations.

Call Semantics

- Parameter passing. Pass parameter values or access path.
- Storage Allocation. Allocate space for non static local variables and bind these variables to the storage.
- Save the caller's execution status, i.e., register values, CPU status bits, and the environment pointer (EP).
- Make sure control is transferred to the callee when entering and back to the caller when exiting.
- Access to non local variable in case of nested subprograms.

Return Semantics

- In case of out mode parameters implemented by *copy*, move the values of associated formal parameters to the actual parameters.
- Return values.
- Deallocate the storage for local variables.
- Restore the execution status of the caller.
- Transfer control back to caller.

Run-time Stack

The *call* and *return* semantics specify that the subprogram last called is the first to complete, i.e., First-In Last-Out (FILO). So it natural to use *STACK*.

Activation Record

Subprogram consists of two separate parts:

1. the actual code of the subprogram, which is fixed, and
2. the non-code, e.g., local variables and data, which may change when the subprogram is executed.

Activation Record Specifies the format, or layout, of the non code part of a subprogram. The form of an activation record is *static*.

Activation Record Instance

Activation Record Instance (ARI) A concrete example of an activation record. During activation, whether recursive or non-recursive, a *new* instance of an activation record is created on the *stack*, including a separate copy of

- the parameters
- local variables, and
- return address.

Outline

Simple Subprogram

- Subprogram cannot be nested. and
- all local variables are *static*.

Call Semantics

- Save the execution status of the caller.
- Parameter Passing. Compute and pass the parameters.
- ~~Space allocation~~.
- Pass the return address to the callee. The return address usually consists of a pointer to the instruction following the call in the code segment of the calling program unit.
- Transfer control to the callee.

Return Semantics

- Pass values to out mode actual parameters.
- Make return values accessible to caller.
- ~~Deallocation~~.
- Restore caller's execution status.
- Transfer control back to caller.

Who Does What – Call

- Save the execution status of the caller.
- Parameter Passing.
- Pass the return address to the callee.
- Transfer control to the callee.

Who Does What – Call

- Save the execution status of the caller.
- Parameter Passing.
- Pass the return address to the callee.
- Transfer control to the callee.

- Either one
- Caller
- Caller
- Caller

Who Does What – Return

- Pass values to out mode parameters.
- Make return values accessible to caller.
- Restore caller's execution status.
- Transfer control back to caller.

Who Does What – Return

- Pass values to out mode parameters.
- Make return values accessible to caller.
- Restore caller's execution status.
- Transfer control back to caller.

- Callee
- Callee
- Either one
- Callee

Linkage Bookkeeping

We need storage for

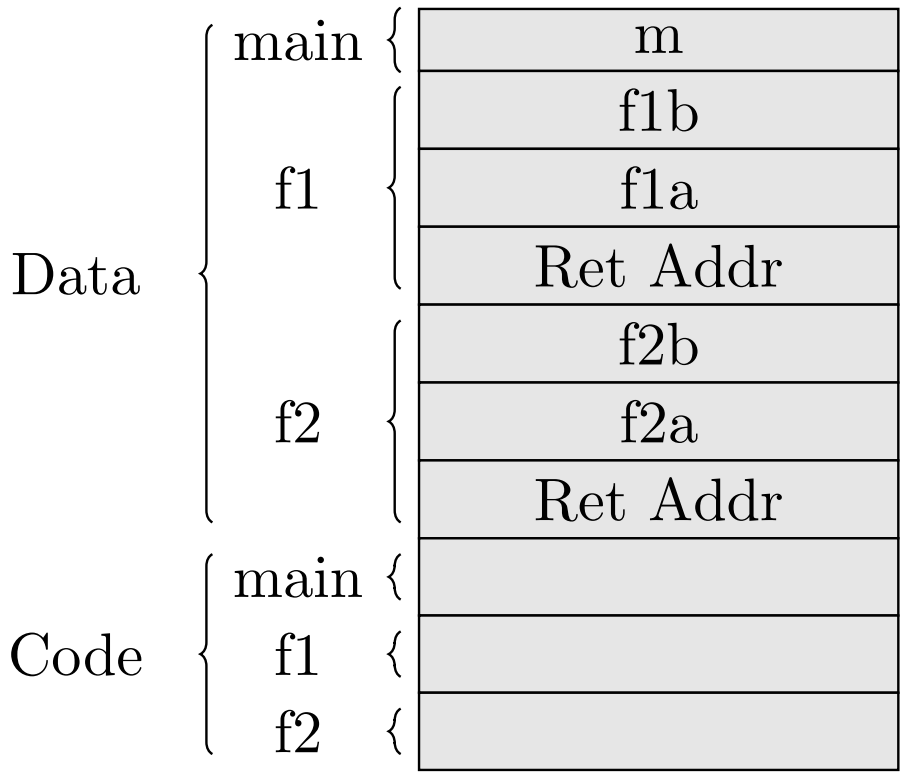
- Status information about the caller
- Parameters
- Return address
- Return value, if any
- Local variables

Activation Record

Local Variable
Parameters
Return Address

```
void f1(int f1a) {
 static int f1b;
}
void f2(int f2a) {
  static int f2b;
}
int main() {
  static int m;
  f1(0);
  f2(0);
}
```

Gathered by *linker*, a.k.a. loader, link editor.



Outline

Non-Simple Subprogram

- Subprogram can ~~not~~ be nested. and
- all local variables are ~~static~~ *stack-dynamic*.

So what?

Non-Simple Subprogram

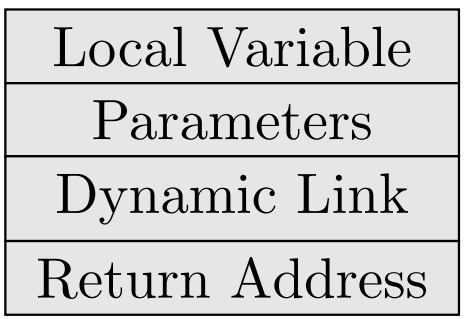
- Subprogram can ~~not~~ be nested. and
- all local variables are ~~static~~ *stack-dynamic*.

So what?

- The compiler must generate code to cause the implicit allocation and deallocation of local variables.
- In case of recursion, there can be more than one instance and each activation requires its activation record instance

Activation Record

The *dynamic link* is a pointer to the base of the activation record instance of the caller.



1. In static scoping, this link is used to provide trace-back information when a run-time error occurs.
2. In dynamic scoping, the dynamic link is used to access non local variables.

EP Pointer

EP, a.k.a. *E*-xtreme stack *P*-ointer, points at the base, or first address of the activation record instance of the main program.

1. When a subprogram is called, the current EP is saved in the new activation record instance as the *dynamic link*.
2. When a subprogram returns,
   - the stack top is set to the EP $-$ 1 and
   - the EP is set to the dynamic link.

Call Semantics

By Caller

- *Create an activation record instance*.
- Save the execution status of the current program unit.
- Compute and pass the parameters.
- Pass the return address to the called.
- Transfer control to the called.

Call Semantics

By Caller

- *Create an activation record instance*.
- Save the execution status of the current program unit.
- Compute and pass the parameters.
- Pass the return address to the called.
- Transfer control to the called.

By Callee

- Save the old EP in the stack as the dynamic link and create the new value.
- Allocate local variables.

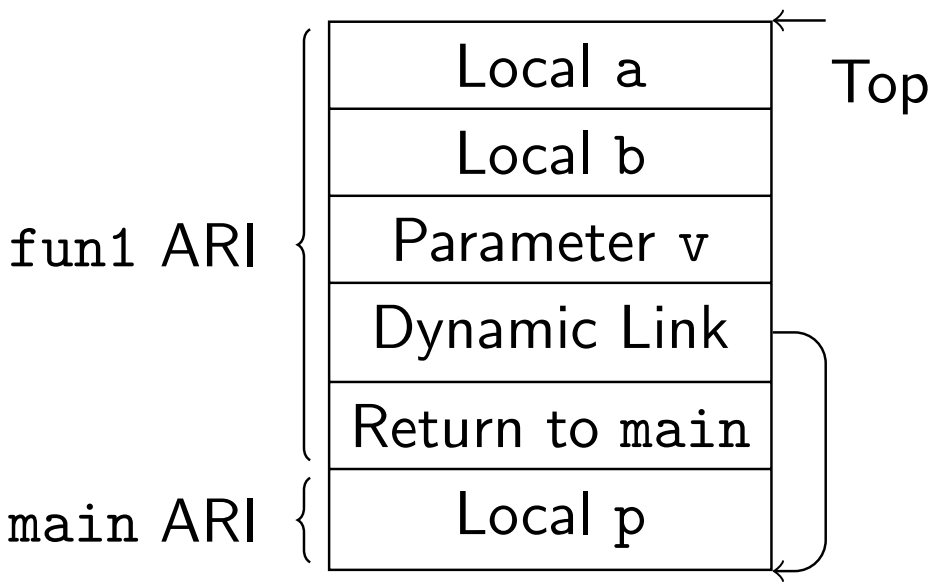
Return Semantics

By Callee

- Copy value back if necessary.
- Make return value, if any, accessible to the caller.
- Set EP to the correct value.
- Restore the execution status of the caller.
- Transfer control back to the caller.

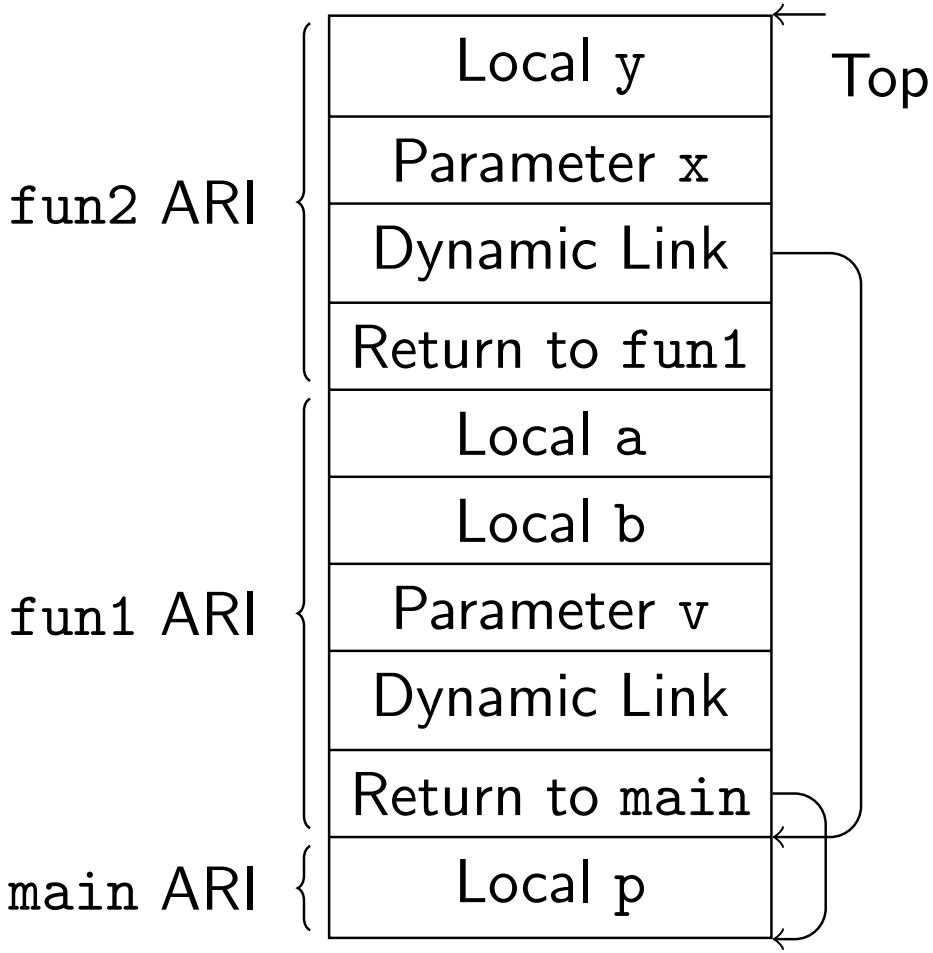
Example – Non-Recursive Calls 1

```
void fun1(float v) {
  int a, b;
  /* [1] */
  fun2(a);
}
void fun2(int x) {
  int y;
  fun3(y);
}
void fun3(int a) {
}
void main() {
  float p;
  fun1(p);
}
```



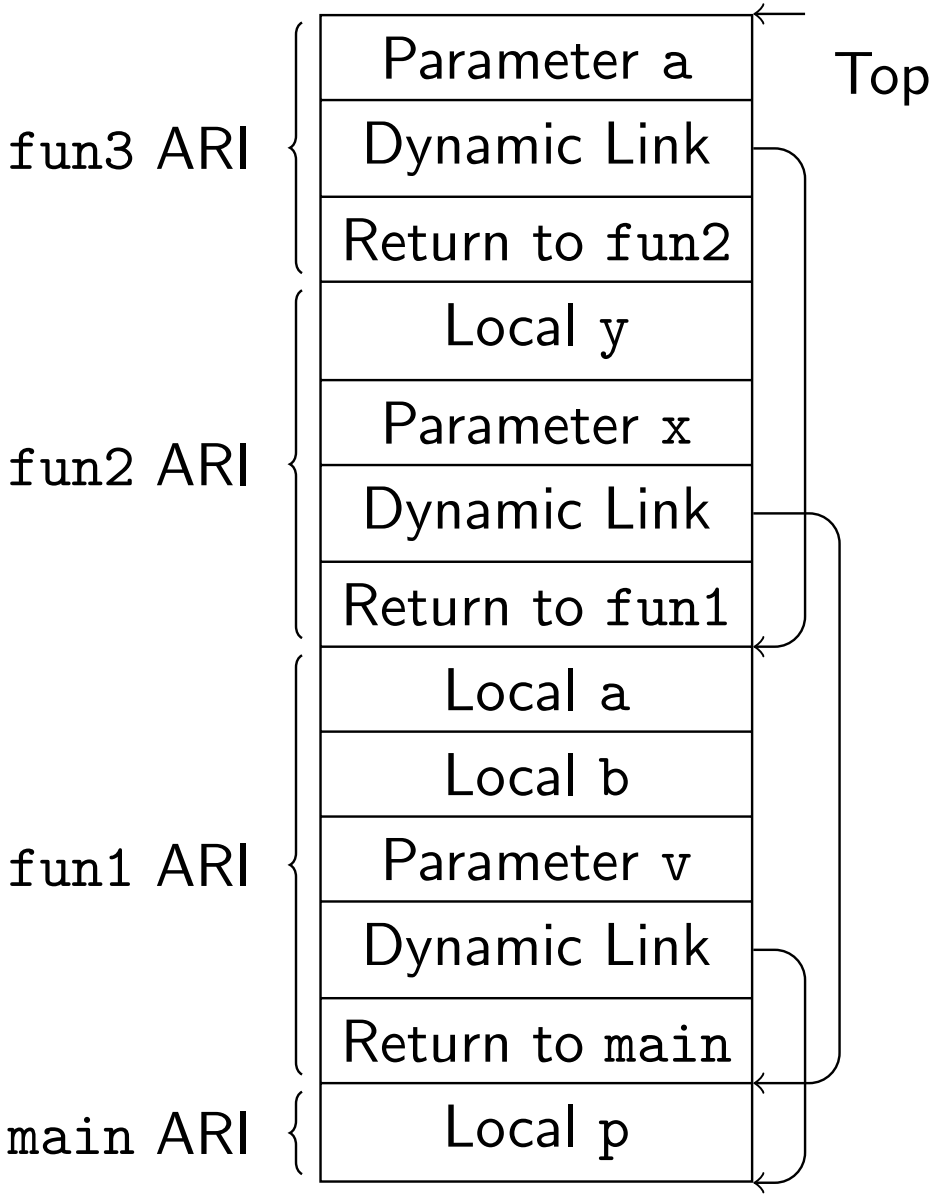
Example – Non-Recursive Calls 2

```
void fun1(float v) {
  int a, b;
  fun2(a);
}
void fun2(int x) {
  int y;
  /* [2] */
  fun3(y);
}
void fun3(int a) {
}
void main() {
  float p;
  fun1(p);
}
```

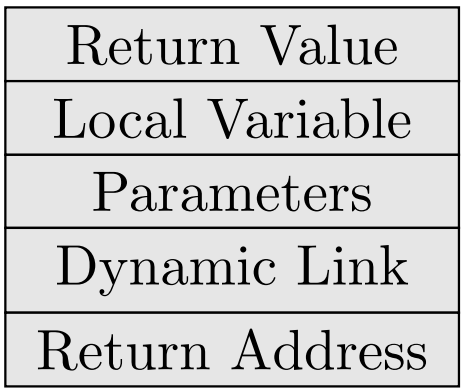


Example – Non-Recursive Calls 3

```
void fun1(float v) {
  int a, b;
  fun2(a);
}
void fun2(int x) {
  int y;
  fun3(y);
}
void fun3(int a) {
  /* [3] */
}
void main() {
  float p;
  fun1(p);
}
```



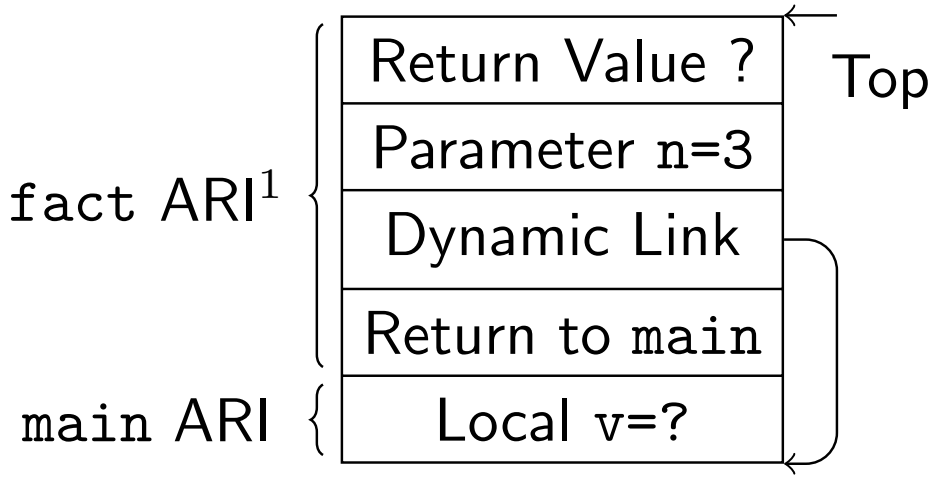
Recursive Subprogram



1. One more field is needed – return value.
2. Local variable and parameter field are both *optional*.

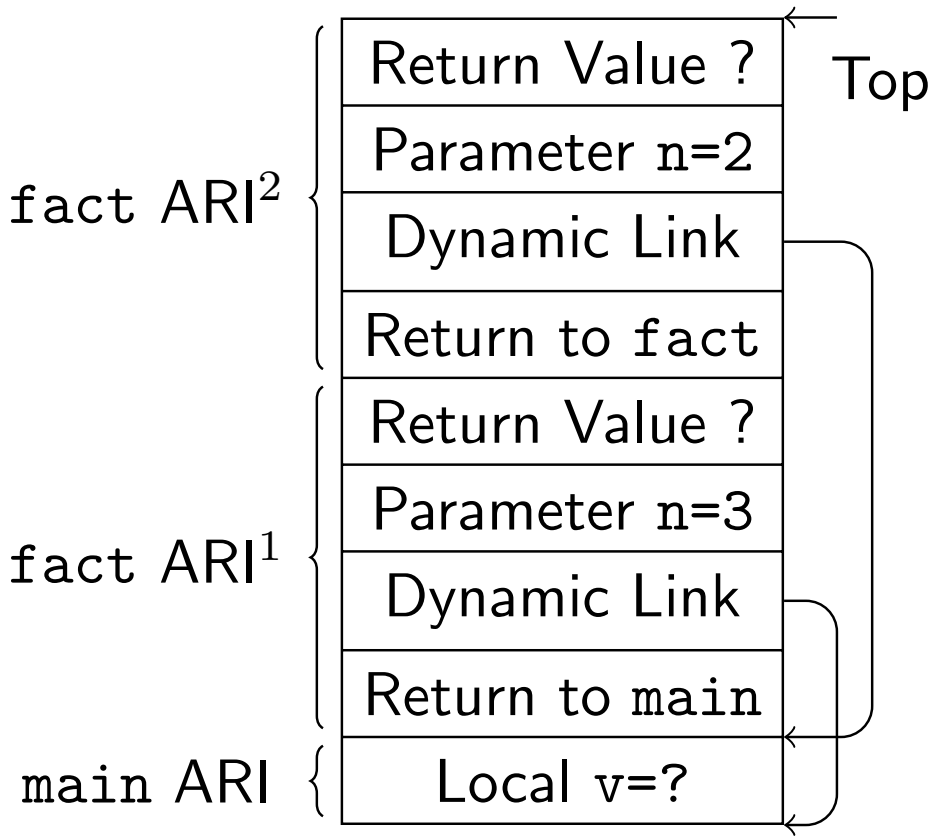
Example – Recursive Calls 1

```c
int fact(int n) {
  /* [1] */
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}

int main() {
  int v = fact(3);
}
```



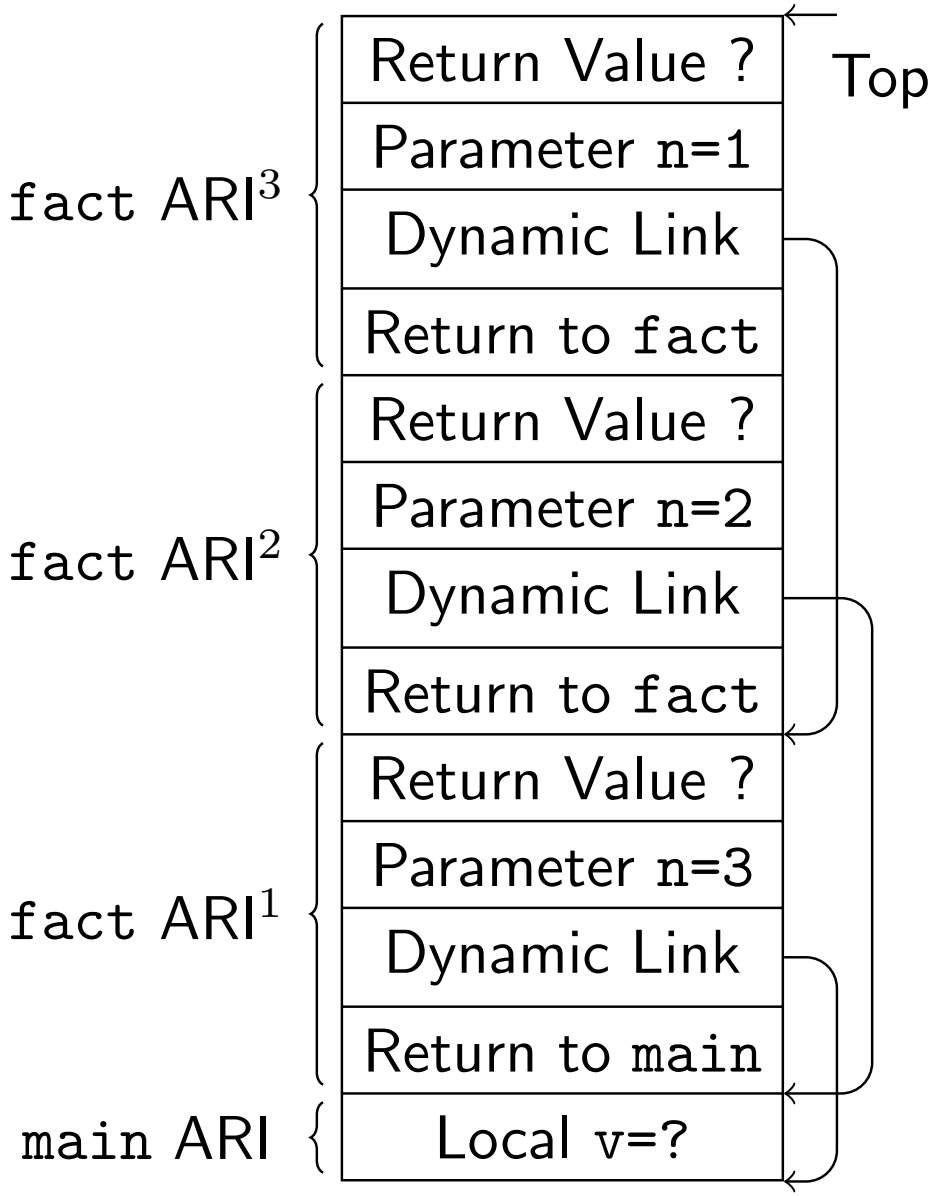
Example – Recursive Calls 1

```c
int fact(int n) {
  /* [1] */
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}

int main() {
  int v = fact(3);
}
```



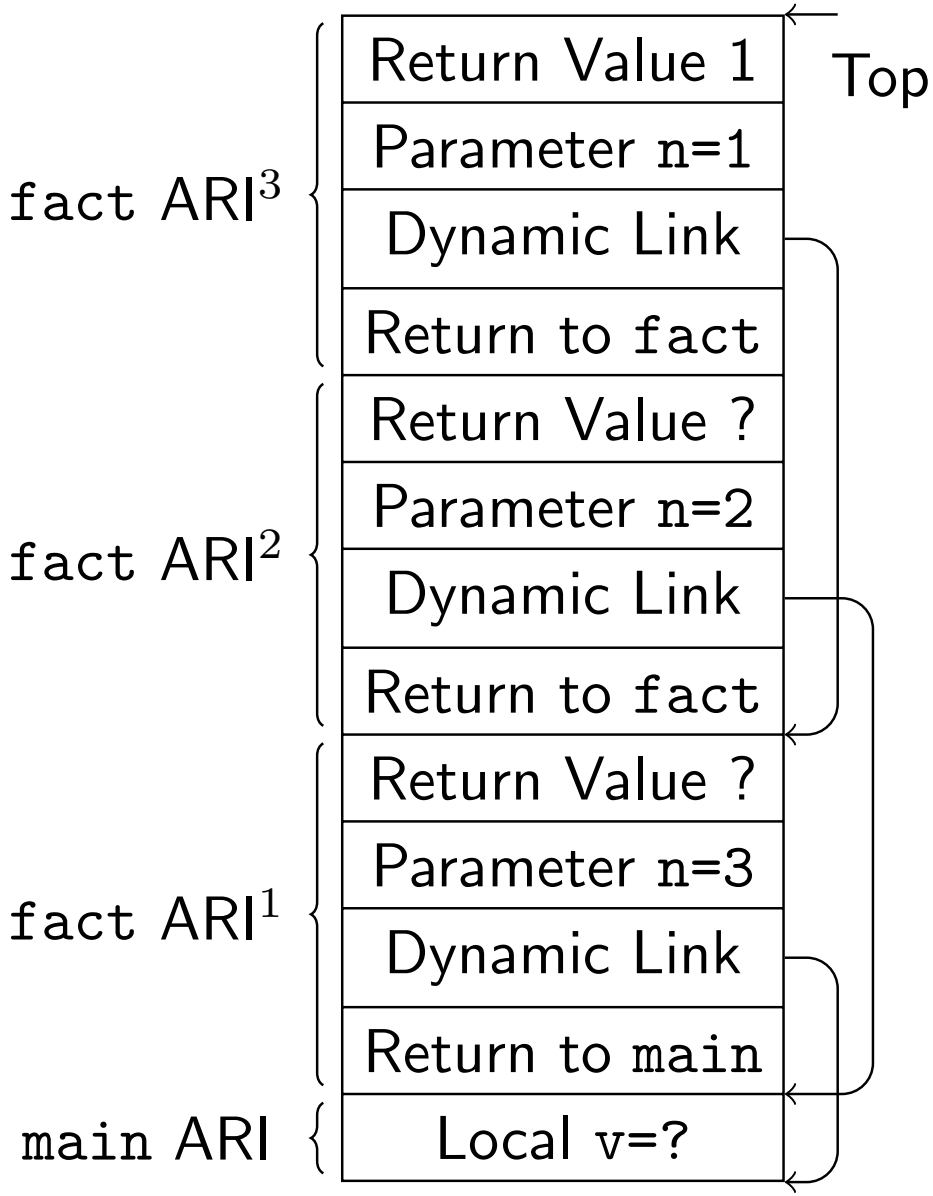
Example – Recursive Calls 1

```
int fact(int n) {
  /* [1] */
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}

int main() {
  int v = fact(3);
}
```



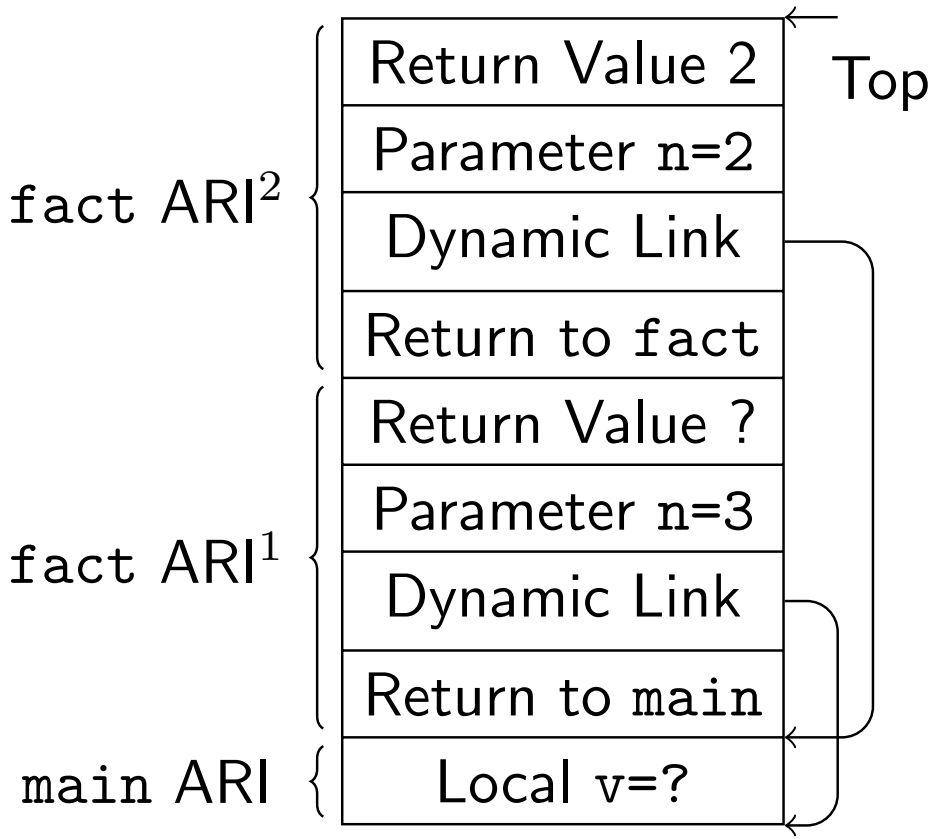
Example – Recursive Calls 2

```
int fact(int n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
  /* [2] */
}

int main() {
  int v = fact(3);
}
```



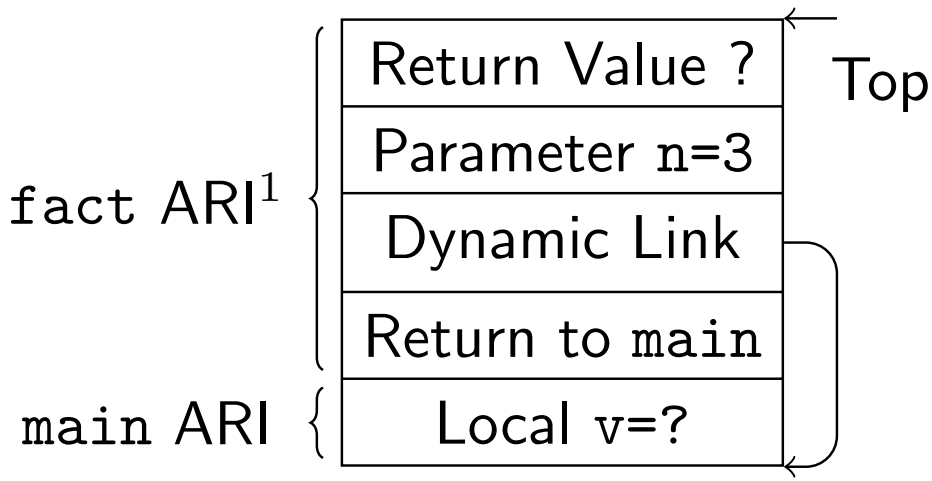
Example – Recursive Calls 2

```
int fact(int n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
  /* [2] */
}

int main() {
  int v = fact(3);
}
```



Example – Recursive Calls 2

```c
int fact(int n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
  /* [2] */
}

int main() {
  int v = fact(3);
}
```



Example – Recursive Calls 3

```c
int fact(int n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}

int main() {
  int v = fact(3);
  /* [3] */
}
```

Outline

Without Nested Subprograms

If subprogram nesting is not allowed, everything is fine. (why??)

Without Nested Subprograms

If subprogram nesting is not allowed, everything is fine. (why??)
Because non local variables are all *global* and resides in a special block called *data segment*.

With Nested Subprograms

In case of nesting, non local variables are accessed in two steps

1. Find the ARI in the stack in which the variable was allocated.
2. Access the variable with the local offset within the ARI.

With Nested Subprograms

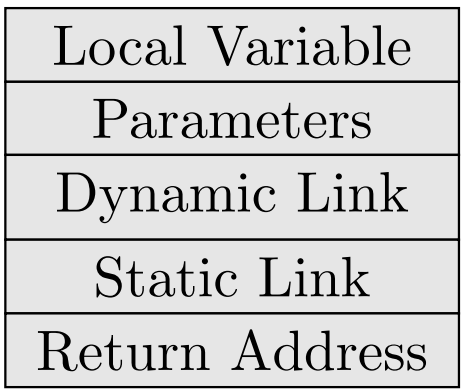
In case of nesting, non local variables are accessed in two steps

1. Find the ARI in the stack in which the variable was allocated.
2. Access the variable with the local offset within the ARI.

Fact: All *non static variables* that can be *non-locally* accessed are in existing ARI and therefore are somewhere in the stack.

Activation Record

The *static link*, a.k.a. static scope pointer, points to the bottom of the activation record instance of an activation of the static parent.

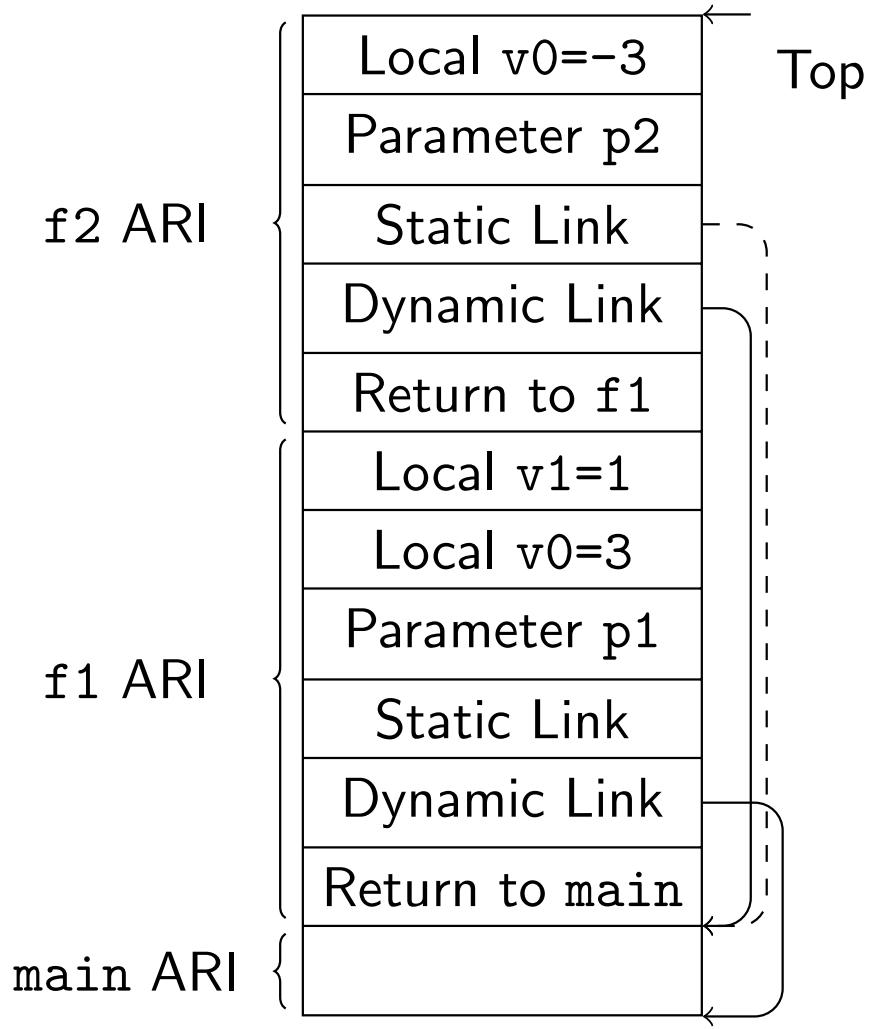


- Static depth, how deeply it is nested.
- Nesting depth, a.k.a., chain offset, difference of static depth.
- (chain offset, local offset)

Example – Nested Subprograms 1

Compile with `-std=c++11`.

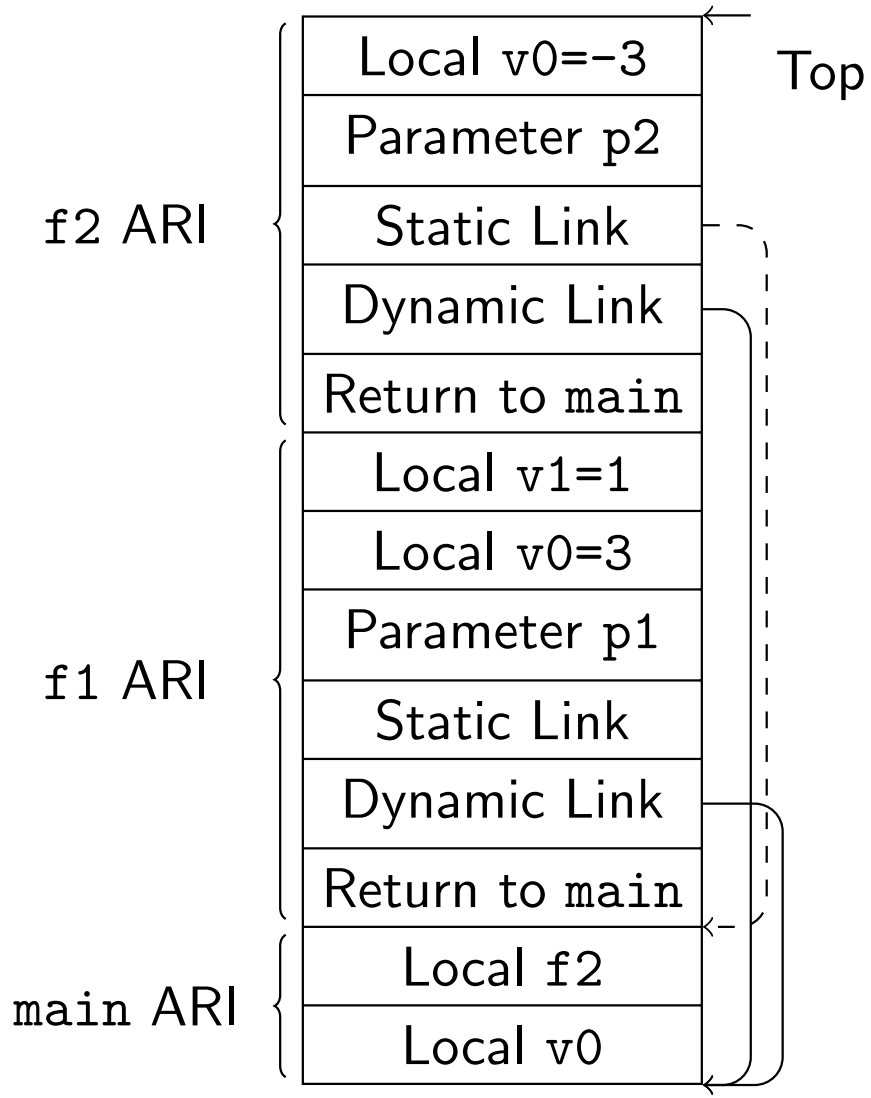
```
#include <cstdio>
void f1(int p1) {
  int v0 = 3;
  int v1 = 1;
  auto f2 = [=](int p2) {
    int v0 = -3;
    // [[1]]
  };
  f2(0);
}
int main() {
  f1(1);
}
```



Example – Nested Subprograms 2

Compile with `-std=c++14`.

```
#include <cstdio>
auto f1(int p1) {
  int v0 = 3;
  int v1 = 1;
  auto f2 = [=](int p2) {
    int v0 = -3;
    // [[1]]
  };
  return f2;
}
int main() {
  int v0 = 33;
  auto f2 = f1(1);
  f2(0);
}
```



Static Chain Maintenance

Both *Call* and *Return*.

- *Return* phase is trivial. Just remove the finished ARI.
- *Call* phase is complex. How to find the static parent ARI?

Static Chain Maintenance

Both *Call* and *Return*.

- *Return* phase is trivial. Just remove the finished ARI.
- *Call* phase is complex. How to find the static parent ARI?
  - Trace the dynamic chain, or

Static Chain Maintenance

Both *Call* and *Return*.

- *Return* phase is trivial. Just remove the finished ARI.
- *Call* phase is complex. How to find the static parent ARI?
  - Trace the dynamic chain, or
  - Whenever compiler notices a function call, it determines the callee's static parent, i.e., the caller's static ancestor. So nesting depth is determined at compile time.

Static Chain Performance

Bad

- Non local variable reference is slower than local.
- Estimate the costs of nonlocal references in a time critical program
- Code refactor may change the static chain.

Good

- Reference to distant non local variables is *rare*.
- In practice, it best.

Limitation and Extension

```
#include <cstdio>
auto f1(int p1) {
  int v0 = 3;
  int v1 = 1;
  auto f2 = [=](int p2) {
    int v0 = -3;
  };
  return f2;
}
int main() {
  int v0 = 33;
  auto f2 = f1(1);
  f2(0);
}
```

Limitation includes

1. Function cannot return function.
2. Function cannot be assigned to variables.

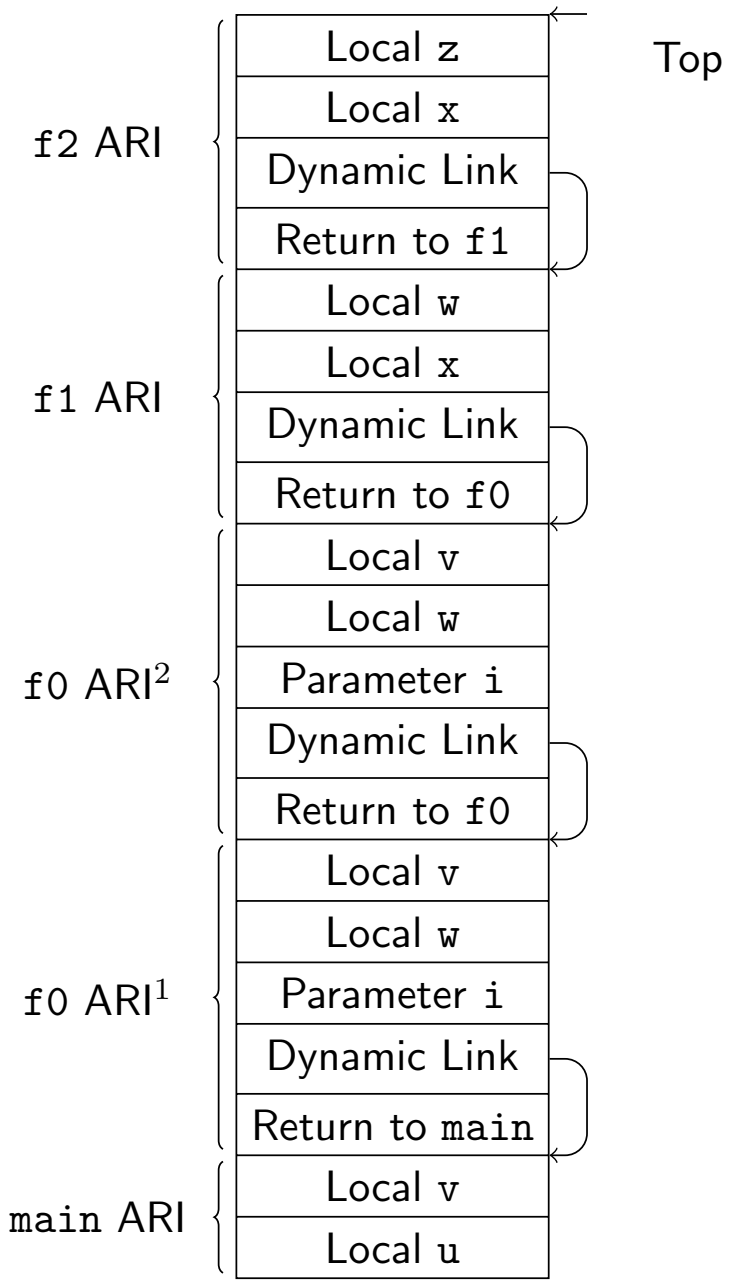
Extension includes

- Non locals (non globals) live both on heap and stack.
- Subprograms need a way to assess non locals on heap.
- Sync non locals living both on heap and stack.

Outline

Deep Access

```
void f2() {
  int z;
  int x = u + v;
}
void f1() {
  int w, x;
  f2();
}
void f0(int i) {
  int v, w;
  if (i == 1) f0(0);
  else f1();
}
int main() {
  int v, u;
  f0(1);
}
```



Shallow Access

```
void f2() {
  int z;
  int x = u + v;
}
void f1() {
  int w, x;
  f2();
}
void f0(int i) {
  int v, w;
  if (i == 1) f0(0);
  else f1();
}
int main() {
  int v, u;
  f0(1);
}
```

- Name stack.
- Central Table.

Outline

Really World

```
int foo(int a1, int a2, int a3,
        int a4, int a5, int a6,
        int a7, int a8, int a9) {
  int local0 = 0x1111;
  int local1 = 0x2222;
  int local2 = 0x3333;
  int local3 = 0x4444;
  int local4 = 0x5555;
  local0 = a1;
  local1 = a2;
  return 0x33;
}
int main() {
  foo(0x1, 0x2, 0x3,
      0x4, 0x5, 0x6,
      0x7, 0x8, 0x9);
  return 0;
}
```

Note that on x86 machines, stack grows *downward* to *lower* memory address.

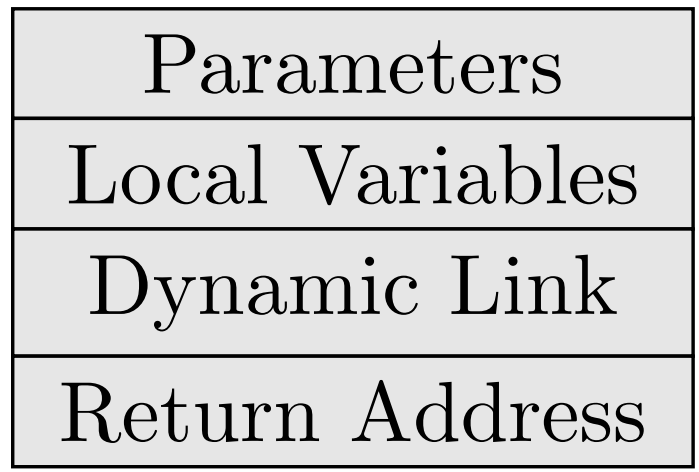


Figure: x86 Linux AR

Summary

- Runtime Stack
- Activation record
- Implementation of static and dynamic scoping