CS 314 Principles of Programming Languages

Lecture 11: Names, Scopes, and Binding

Prof. Zheng Zhang

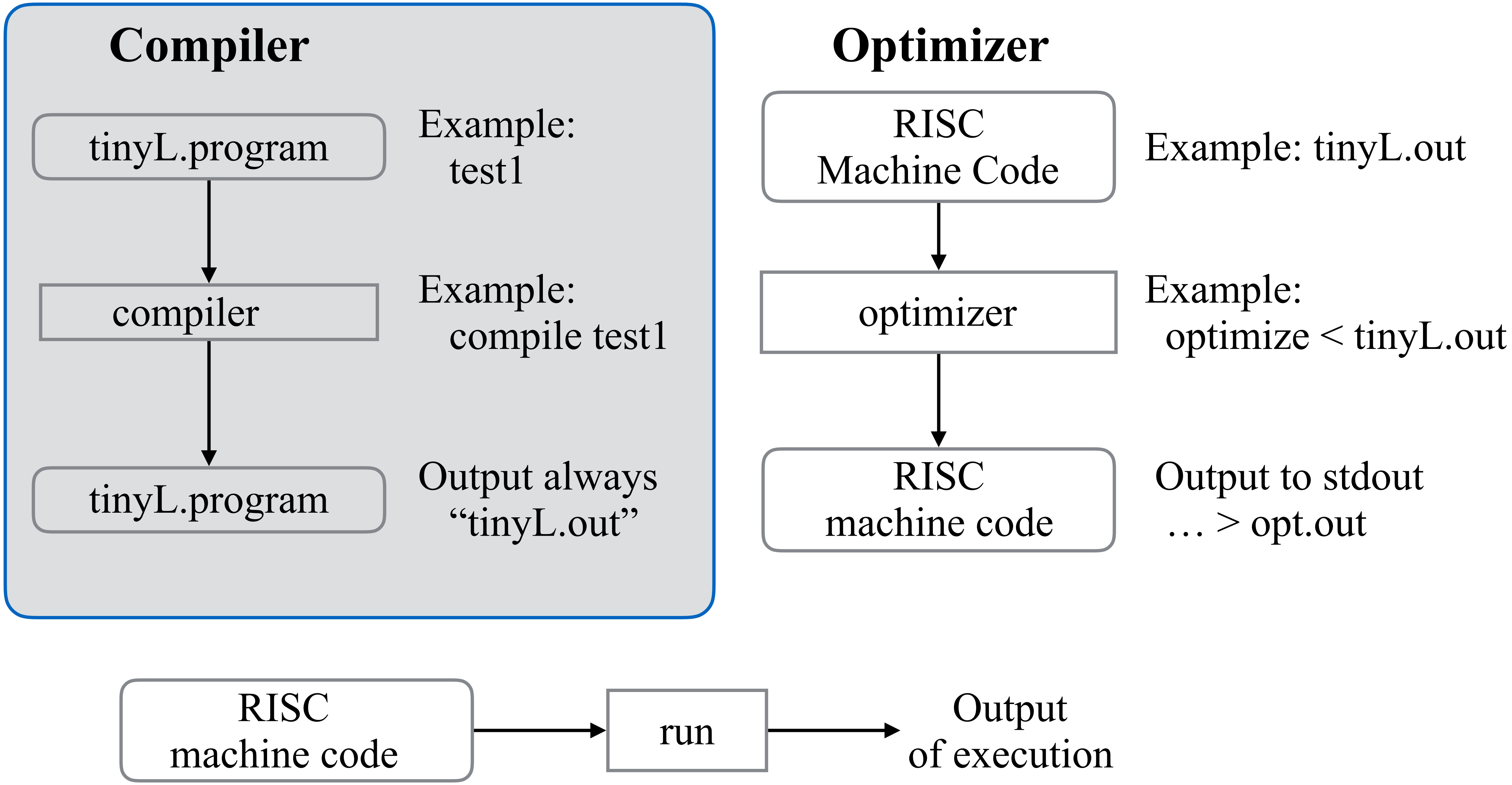
Rutgers University

October 10, 2018

Class Information

- Project 1 posted (open at noon), due Tuesday 10/23 11:55 pm EDT.
- Midterm exam will be on 11/7 Wednesday, in class, closed-book.
- Project 2 will be released immediately after midterm exam.
- My office hour this week is changed to Thursday 4:00pm-5:00pm.

Project 1: overview



Compiler

tinyL.program

Example:
test1

compiler

Example:
compile test1

tinyL.program

Output always
“tinyL.out”

Optimizer

RISC
Machine Code

Example: tinyL.out

optimizer

Example:
optimize < tinyL.out

RISC
machine code

Output to stdout
… > opt.out

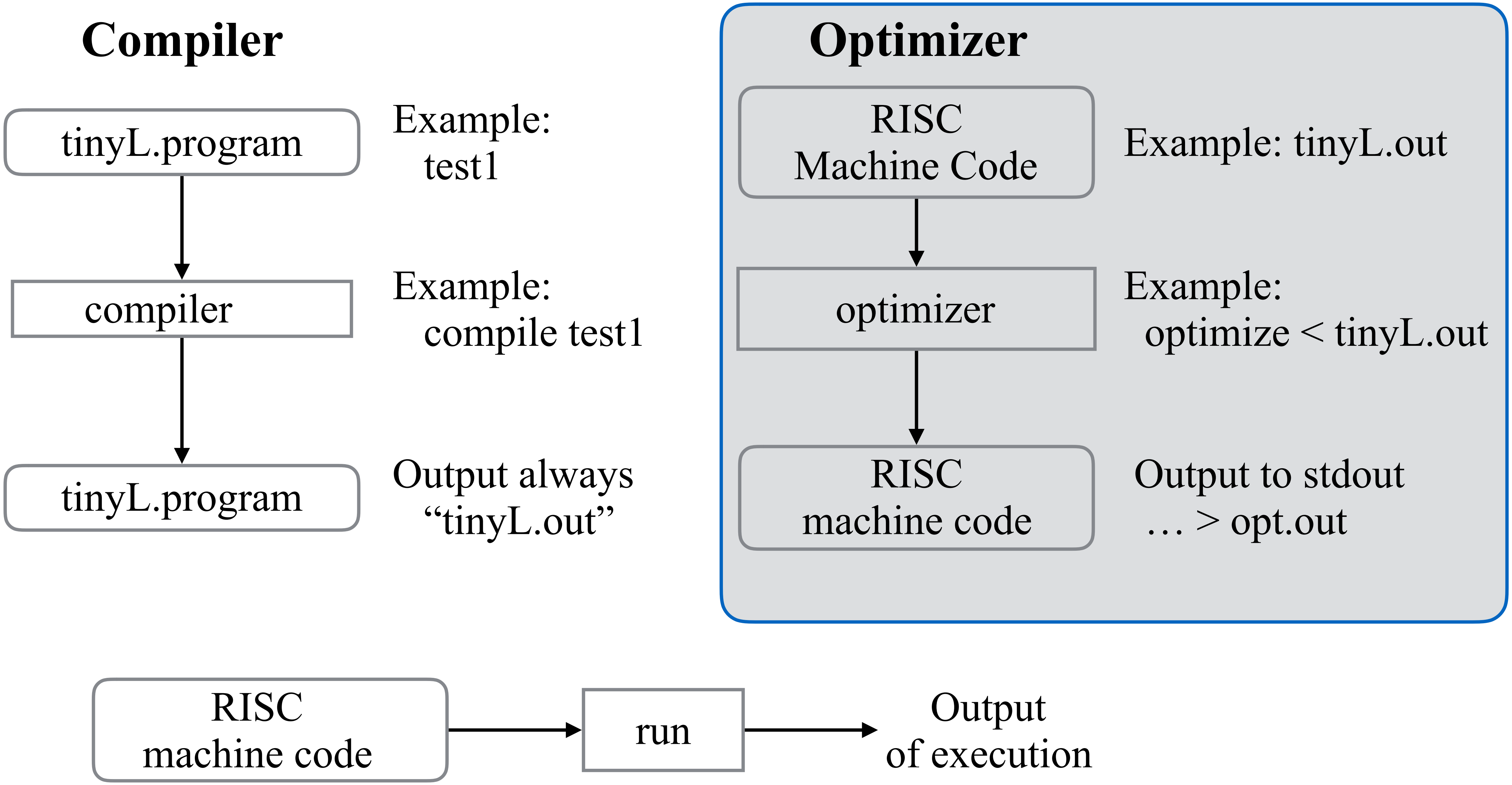
RISC
machine code

run

Output
of execution

Example: run opt.out

Project 1: overview



Compiler

tinyL.program

compiler

tinyL.program

Example:
test1

Example:
compile test1

Output always
“tinyL.out”

Optimizer

RISC
Machine Code

optimizer

RISC
machine code

Example: tinyL.out

Example:
optimize < tinyL.out

Output to stdout
… > opt.out

RISC
machine code

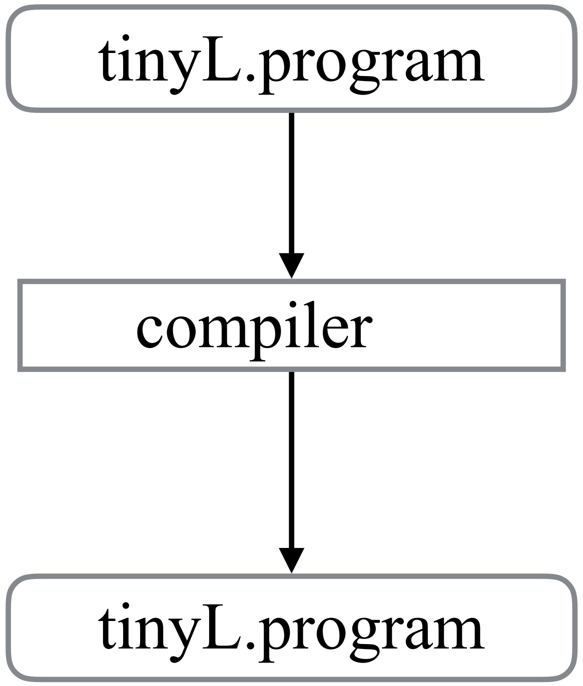
run

Output
of execution

Example: run opt.out

Project 1: overview

Compiler


tinyL.program

compiler

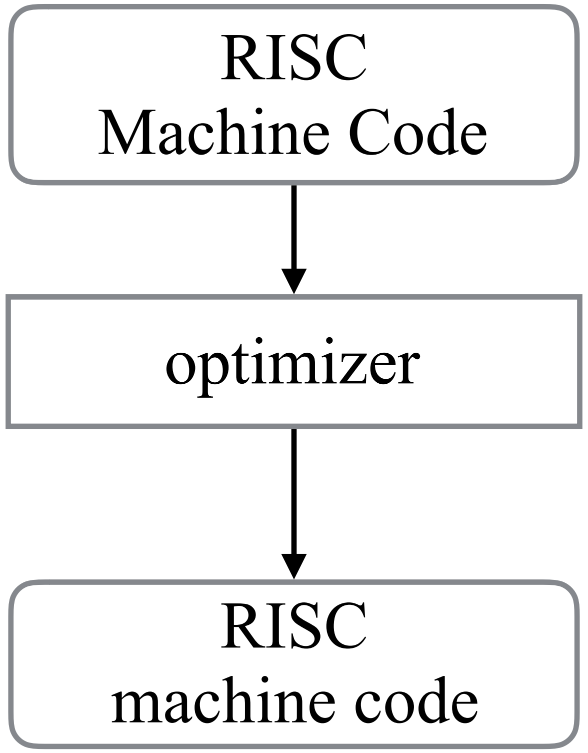
tinyL.program

Example:
test1

Example:
compile test1

Output always
“tinyL.out”

Optimizer


RISC
Machine Code

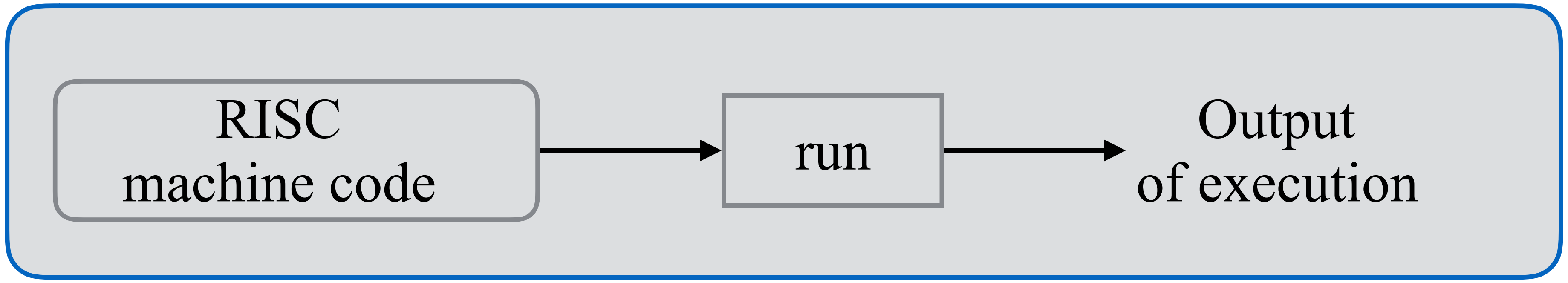
optimizer

RISC
machine code

Example: tinyL.out

Example:
optimize < tinyL.out

Output to stdout
… > opt.out


RISC
machine code

run

Output
of execution

Example: run opt.out

Project 1 Part II: Constant Propagation

Constant Propagation: Substitute the values of known constants in expressions at compile time. Fold multiple instructions into one if necessary. The constant values might "propagate" and require multiple passes of analysis.

Example:

Original Code

```
LOADI Ra #1
LOADI Rb #1
ADD Rc Ra Rb
LOADI Rd #2
LOADI Re #2
ADD Rf Re Rd
ADD Rg Rf Rc
```

After One Pass

```
LOADI Rc #2
LOADI Rf #4
ADD Rg Rc Rf
```

See project description for more details.

Project 1 Part II: Constant Propagation

Constant Propagation: Substitute the values of known constants in expressions at compile time. Fold multiple instructions into one if necessary. The constant values might “propagate” and require multiple passes of analysis.

Example:

Original Code

```
LOADI Ra #1
LOADI Rb #1
ADD Rc Ra Rb
LOADI Rd #2
LOADI Re #2
ADD Rf Re Rd
ADD Rg Rf Rc
```

After One Pass

```
LOADI Rc #2
LOADI Rf #4
ADD Rg Rc Rf
```

See project description for more details.

Project 1 Part II: Constant Propagation

Constant Propagation: Substitute the values of known constants in expressions at compile time. Fold multiple instructions into one if necessary. The constant values might "propagate" and require multiple passes of analysis.

Example:

Original Code

```
LOADI Ra #1
LOADI Rb #1
ADD Rc Ra Rb
LOADI Rd #2
LOADI Re #2
ADD Rf Re Rd
ADD Rg Rf Rc
```

After One Pass

```
LOADI Rc #2
LOADI Rf #4
ADD Rg Rc Rf
```

Is this good enough?

See project description for more details.

Project 1 Part II: Constant Propagation

Constant Propagation: Substitute the values of known constants in expressions at compile time. Fold multiple instructions into one if necessary. The constant values might "propagate" and require multiple passes of analysis.

Example:

Original Code

```
LOADI Ra #1
LOADI Rb #1
ADD Rc Ra Rb
LOADI Rd #2
LOADI Re #2
ADD Rf Re Rd
ADD Rg Rf Rc
```

After One Pass

```
LOADI Rc #2
LOADI Rf #4
ADD Rg Rc Rf
```

After Another Pass

```
LOADI Rg #6
```

See project description for more details.

Names, Bindings, and Scope

What’s a name?

A name is a mnemonic character string used to represent something else.

What’s in a name?

- Has associated “attributes”
  Examples: type, memory location, read/write permission, storage class, access restrictions.
- Has a meaning
  Examples: represents a semantic object, a type description, an integer value, a function implementation, a memory address.

Names, Bindings, and Scope

Bindings – association of a name with the thing it "names"

- **Compile time**: during compilation process - static (e.g.: macro expansion, type definition)
- **Link time**: separately compiled modules/files are joined together by the linker (e.g: adding the standard library routines for I/O (stdio.h), external variables)
- **Run time**: when program executes - dynamic

Binding Time - Choices

- **Early binding** times — more efficient (faster) at run time
- **Late binding** times — more flexible (postpone binding decision until more “information” is available)
- Examples of static binding (early):
  - functions in C
  - types in C
- Examples of dynamic binding (late):
  - virtual methods in Java
  - dynamic typing in Javascript, Scheme

Note: dynamic linking is somewhat in between static and dynamic binding; the function signature has to be known (static), but the implementation is linked and loaded at run time (dynamic).

How to Maintain Bindings

- **Symbol table**: maintained by compiler during compilation

  names ⇒ *attributes*

- **Referencing Environment**:

  maintained by compiler-generated-code during program execution

  names ⇒ *memory locations*

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

local variable, procedure def.

```
program L;
          var n: char;           {n declared in L}
          procedure W;
          begin
                    write (n);   {n referenced in W}
          end;
          procedure D;
                    var n: char; {n declared in D}
          begin
                    n := ‘D’;    {n referenced in D}
                    W
          end;
```

implementation

```
begin
          n := ‘L’;              {n referenced in L}
          W;
          D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;            {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := ‘D’;       {n referenced in D}
                W
        end;
begin
        n := ‘L’;               {n referenced in L}
        W;
        D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;                {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := 'D';       {n referenced in D}
                W
        end;
begin
        n := 'L';                   {n referenced in L}
        W;
        D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;            {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := ‘D’;       {n referenced in D}
                W
        end;
begin
        n := ‘L’;               {n referenced in L}
        W;
        D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;            {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := ‘D’;       {n referenced in D}
                W
        end;
begin
        n := ‘L’;               {n referenced in L}
        W;
        D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;            {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := ‘D’;       {n referenced in D}
                W
        end;
begin
        n := ‘L’;               {n referenced in L}
        W;
        D
end
```

Scope Example

Nested Subroutines (Algol 60, Ada, Common Lisp, Python, ….)

```
program L;
        var n: char;            {n declared in L}
        procedure W;
        begin
                write (n);      {n referenced in W}
        end;
        procedure D;
                var n: char;    {n declared in D}
        begin
                n := 'D';       {n referenced in D}
                W
        end;
begin
        n := 'L';               {n referenced in L}
        W;
        D
end
```

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$$L \Rightarrow W$$

$$L \Rightarrow D \Rightarrow W$$

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$L \Rightarrow W$

$L \Rightarrow D \Rightarrow W$

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

Calling Chain:

$L \Rightarrow W$

$L \Rightarrow D \Rightarrow W$

```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

Which “n”?

```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := ‘D’; {n referenced in D}
                W
        end;
begin
        n := ‘L’;   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$L \Rightarrow W$

$L \Rightarrow D \Rightarrow W$

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

L

Which “n”?

```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := ‘D’; {n referenced in D}
                W
        end;
begin
        n := ‘L’;   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$$L \Rightarrow W$$

$$L \Rightarrow D \Rightarrow W$$

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

L

Calling Chain:

$L \Rightarrow W$

$L \Rightarrow D \Rightarrow W$

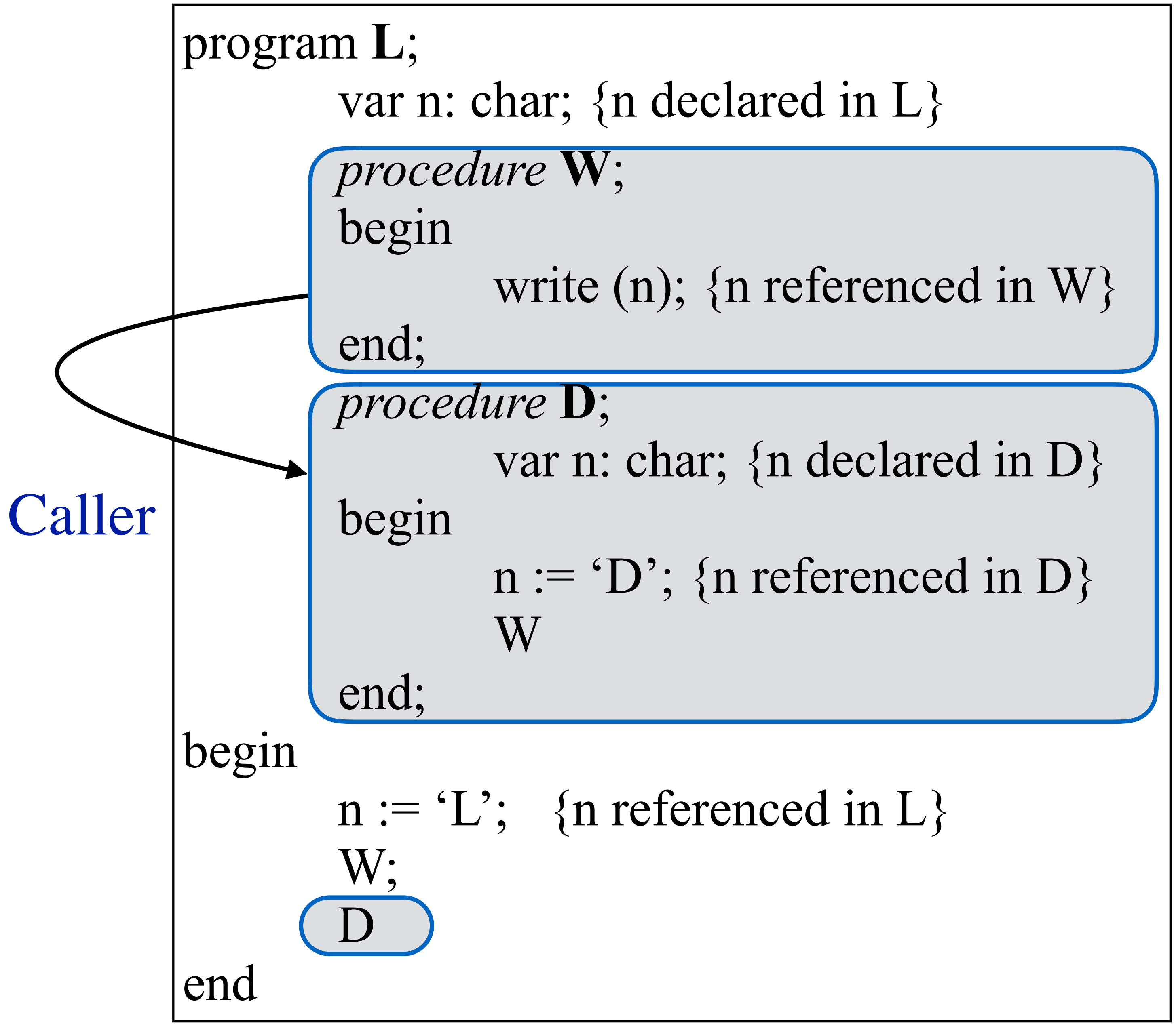
```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

L



```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$$L \Rightarrow W$$

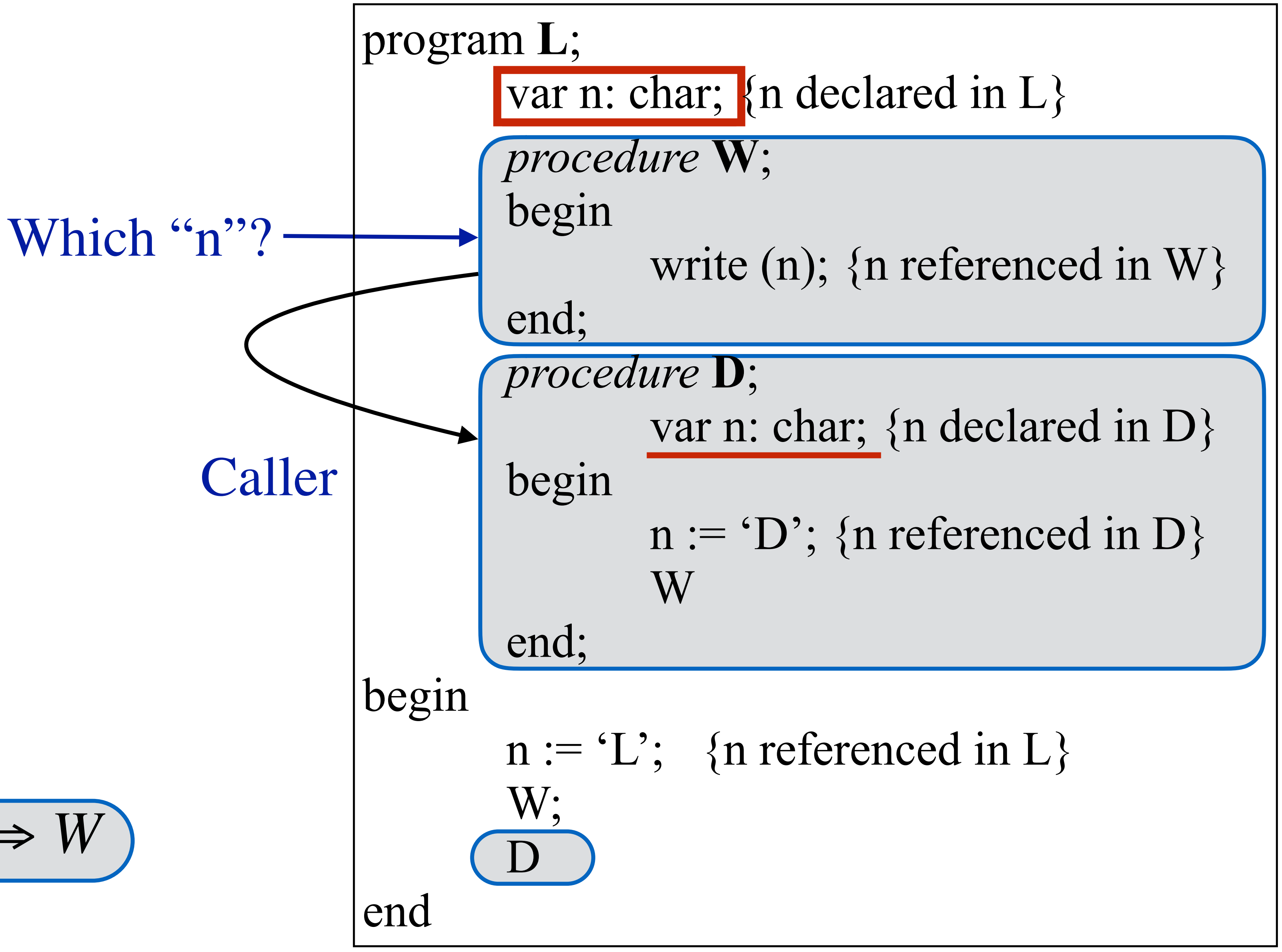
$$\boxed{L \Rightarrow D \Rightarrow W}$$

Lexical Scope

- Non-local variables are associated with declarations at ***compile*** time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

The output is ?

L
L



```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$L \Rightarrow W$

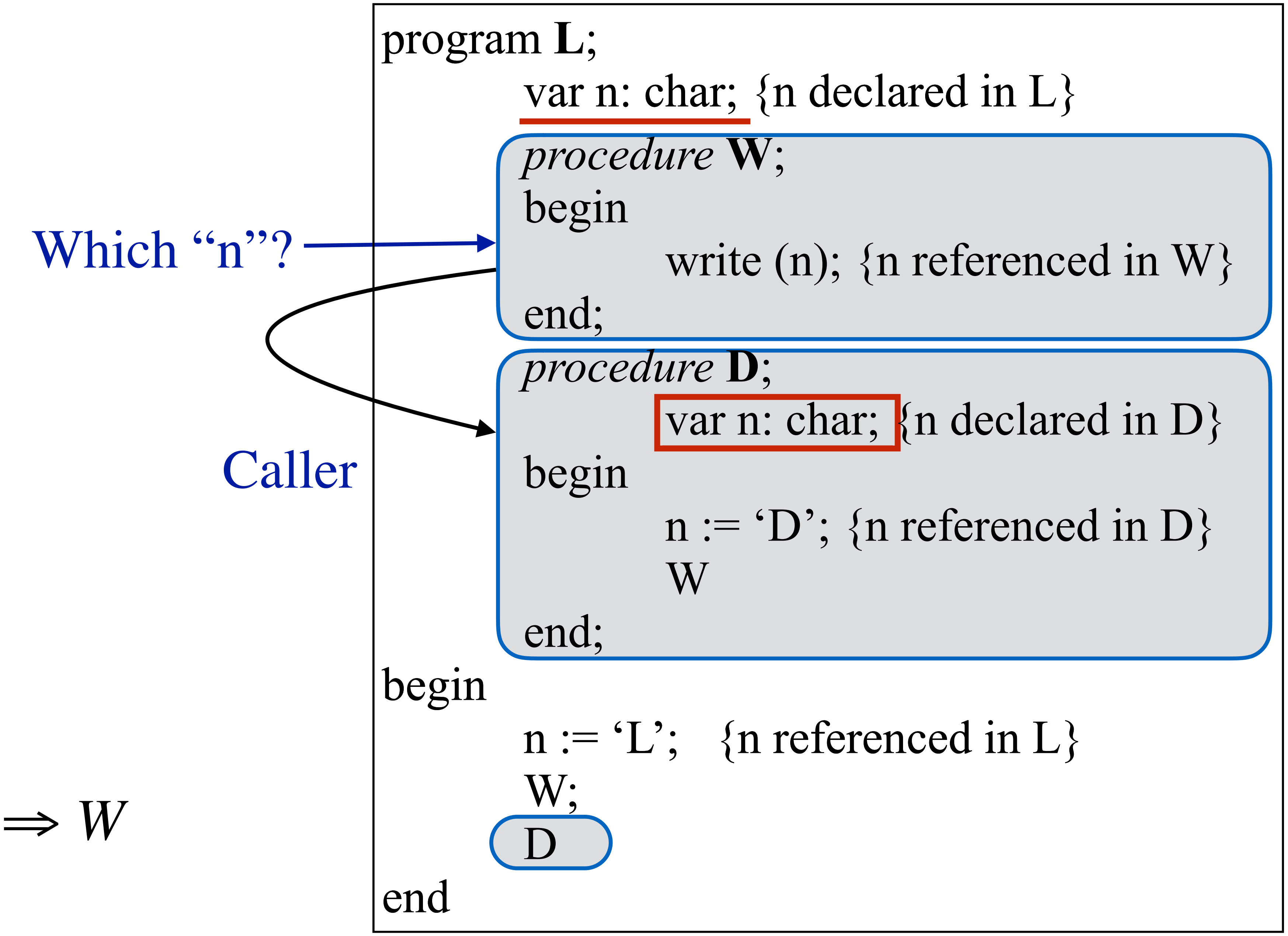
$L \Rightarrow D \Rightarrow W$

Dynamic Scope

- Non-local variables are associated with declarations at ***run*** time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable

The output is ?

L
D



```
program L;
        var n: char; {n declared in L}
        procedure W;
        begin
                write (n); {n referenced in W}
        end;
        procedure D;
                var n: char; {n declared in D}
        begin
                n := 'D'; {n referenced in D}
                W
        end;
begin
        n := 'L';   {n referenced in L}
        W;
        D
end
```

Calling Chain:

$$L \Rightarrow W$$

$$L \Rightarrow D \Rightarrow W$$

Lexical Scope v.s. Dynamic Scope

Lexical Scope

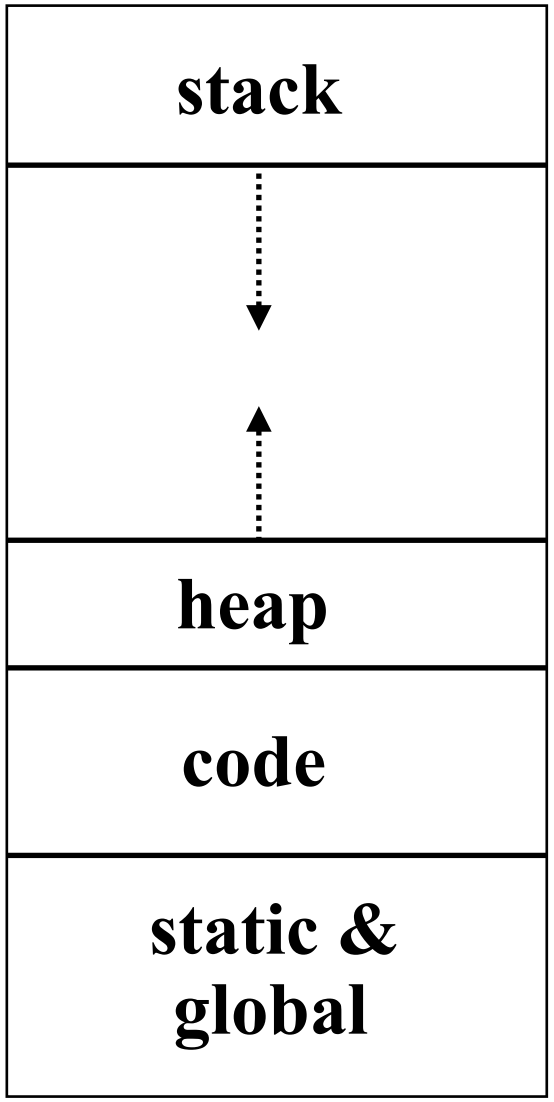
- Non-local variables are associated with declarations at *compile* time
- Find the smallest block ***syntactically*** enclosing the reference and containing a declaration of the variable

Dynamic Scope

- Non-local variables are associated with declarations at *run* time
- Find the ***most recent, currently*** active run-time stack frame containing a declaration of the variable

Review: Program Memory Layout

- Static objects are given an absolute address that is retained throughout the execution of the program
- Stack objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns
- Heap objects are allocated and deallocated at any arbitrary time



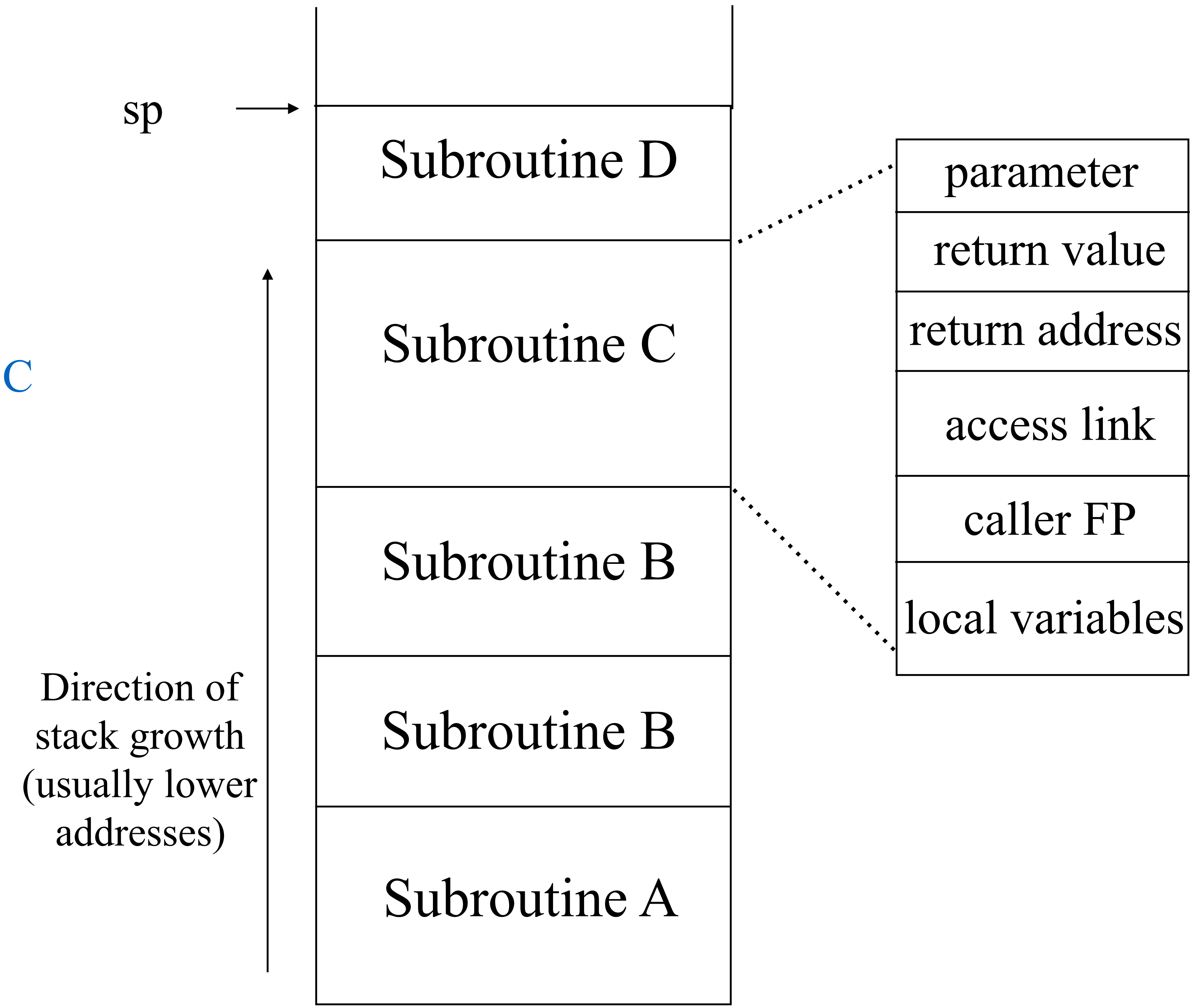
Procedure Activations

- Begins when control enters activation (call)
- Ends when control returns from call

Calling chain: $A \Rightarrow B \Rightarrow B \Rightarrow C \Rightarrow D$

Example:

```
procedure C:
    D
procedure B:
    if…then B else C
procedure A:
    B
main program:
    A
```



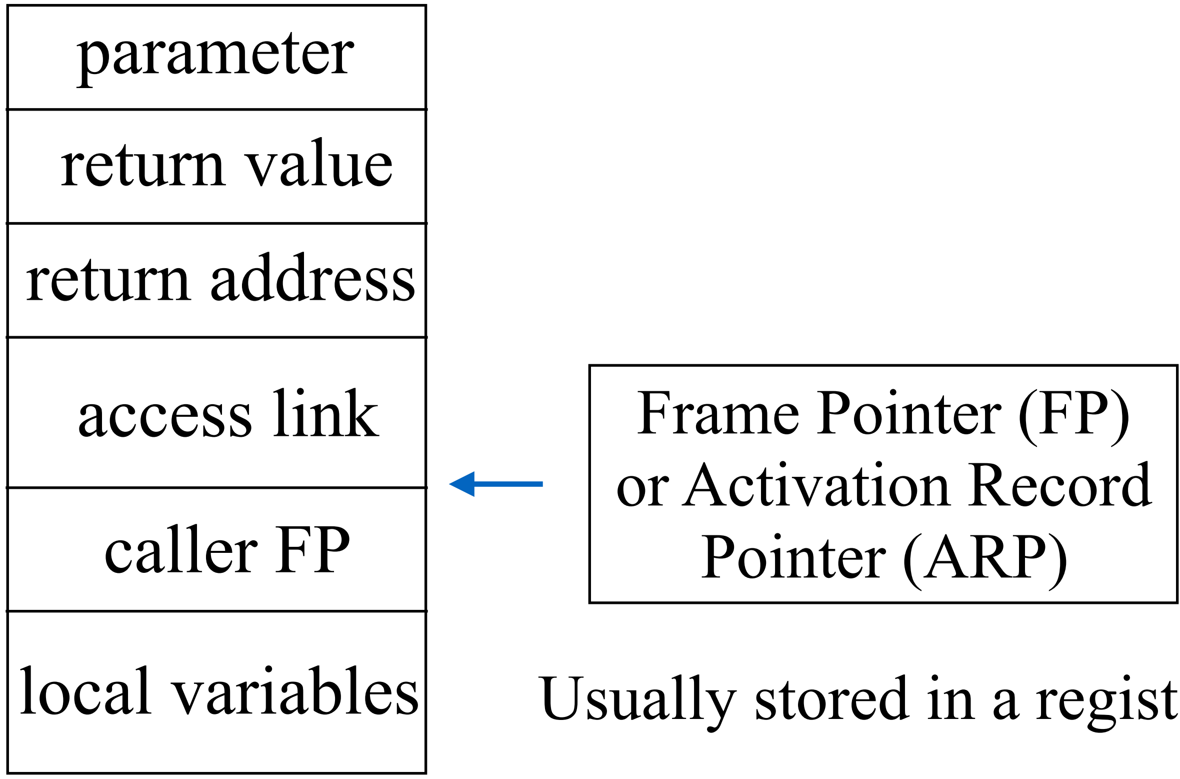
Procedure Activations

- Run-time stack contains frames from main program & active procedure
- Each **stack frame** includes:
  1. Pointer to stack frame of caller
     (**control link** for stack maintenance and dynamic scoping)
  2. Return address (within calling procedure)
  3. Mechanism to find non-local variables (**access link** for lexical scoping)
  4. Storage for parameters, local variables and final values
  5. Other temporaries including intermediate values & saved register

Stack Frame

or

Activation Record



parameter

return value

return address

access link

caller FP

local variables

Frame Pointer (FP) or Activation Record Pointer (ARP)

Usually stored in a register

Lexical Scoping and Dynamic Scoping Implementation

How do we look for non-local variables?

```
Program
    x, y: integer   // declarations of x and y
    Procedure B     // declaration of B
        y, z: real  // declaration of y and z
    begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
    end
    Procedure C     // declaration of C
        x: real
    begin
                ...
        call B // occurrence of B
    end
begin
    ...
    call C     // occurrence of C
    call B     // occurrence of B
end
```

Lexical Scoping and Dynamic Scoping Implementation

How do we look for non-local variables?

```
Program
    x, y: integer   // declarations of x and y
    Procedure B     // declaration of B
        y, z: real  // declaration of y and z
    begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
    end
    Procedure C     // declaration of C
        x: real
    begin
                    ...
        call B // occurrence of B
    end
begin
    ...
    call C    // occurrence of C
    call B    // occurrence of B
end
```

Lexical Scoping and Dynamic Scoping Implementation

How do we look for non-local variables?

```
Program
    x, y: integer   // declarations of x and y
    Procedure B     // declaration of B
        y, z: real  // declaration of y and z
    begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
    end
    Procedure C     // declaration of C
        x: real
    begin
                ...
        call B // occurrence of B
    end
begin
    ...
    call C     // occurrence of C
    call B     // occurrence of B
end
```

Lexical Scoping and Dynamic Scoping Implementation

How do we look for non-local variables?

```
Program
    x, y: integer   // declarations of x and y
    Procedure B     // declaration of B
        y, z: real  // declaration of y and z
    begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
    end
    Procedure C     // declaration of C
        x: real
    begin
                ...
        call B // occurrence of B
    end
begin
    ...
    call C     // occurrence of C
    call B     // occurrence of B
end
```

Lexical Scoping and Dynamic Scoping Implementation

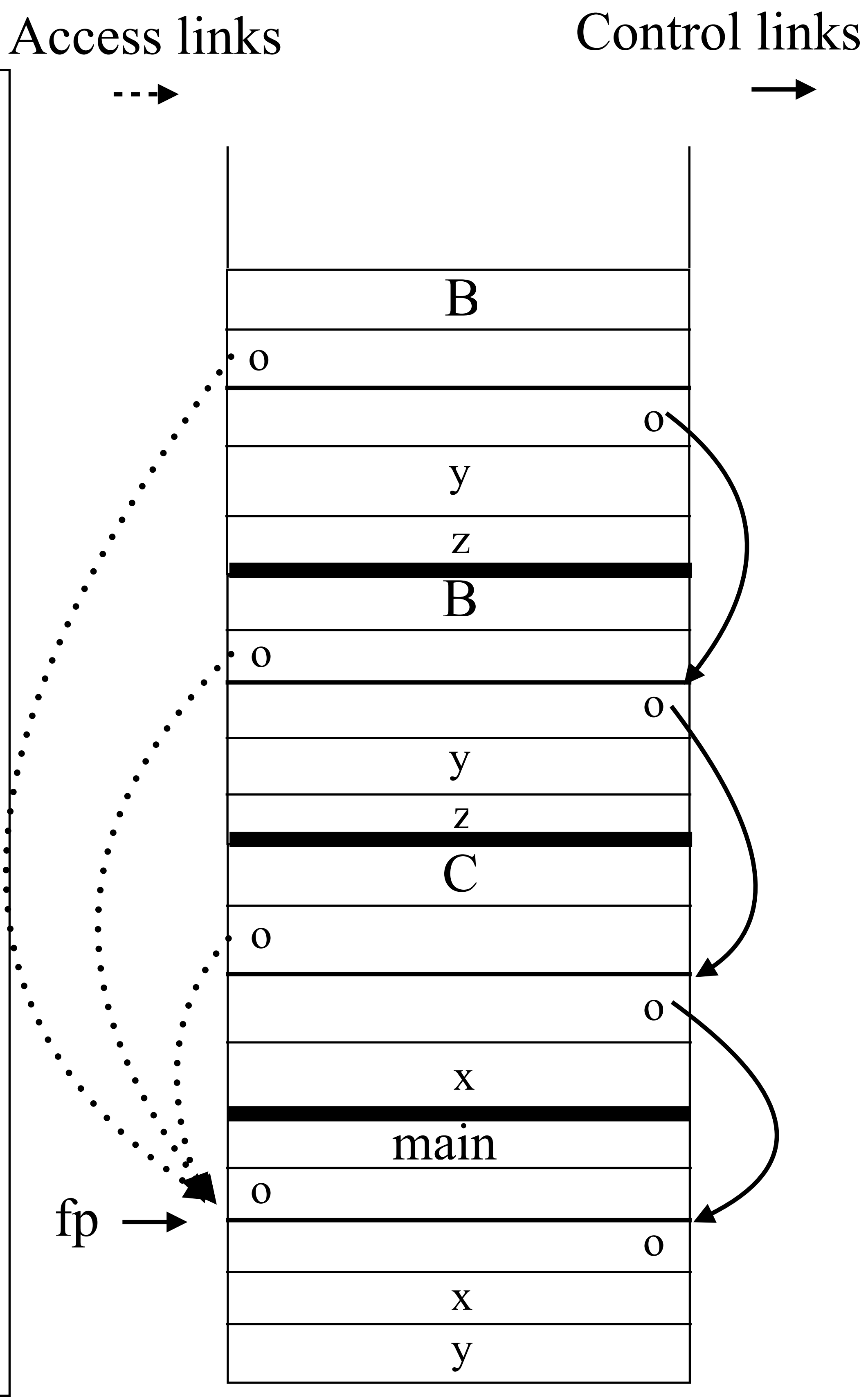
How do we look for non-local variables?

```
Program
    x, y: integer   // declarations of x and y
    Procedure B     // declaration of B
        y, z: real  // declaration of y and z
    begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
    end
    Procedure C     // declaration of C
        x: real
    begin
                ...
        call B // occurrence of B
    end
begin
    ...
    call C     // occurrence of C
    call B     // occurrence of B
end
```

Lexical Scoping and Dynamic Scoping Example

Calling chain: MAIN⇒ C ⇒ B ⇒ B

```
Program
        x, y: integer   // declarations of x and y
        Procedure B     // declaration of B
                y, z: real  // declaration of y and z
        begin
                ...
                y = x + z // occurrences of y, x, and z
                if (...) call B // occurrence of B
        end
         Procedure C    // declaration of C
                x: real
        begin
                ...
                call B // occurrence of B
        end
begin
         ...
         call C     // occurrence of C
         call B     // occurrence of B
end
```



Look up Non - local Variable Reference

Access links and **control links** are used to look for non-local variable references.

Static Scope:

Access link points to the stack frame of the ***most recently*** *activated lexically enclosing procedure*

⇒ Non-local name binding is *determined* at *compile time*, and *implemented* at *run-time*

Dynamic Scope:

Control link points to the stack frame of ***caller***

⇒ Non-local name binding is *determined* and *implemented* at *run-time*

Access to Non-Local Data

How does the code find non-local data at run-time?

Real globals:

- visible everywhere
- translated into a logical address at compile time

Lexical scoping:

- view variables as (level, offset) pairs, (**compile-time symbol table**)
- use (level, offset) pair to get address by using chains of access link (at **run-time**)

Dynamic scoping:

- variable names are preserved
- look-up of variable name uses chains of control links (at **run-time**)

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
 x, y: integer   // declarations of x and y
 Procedure B     // declaration of B
    y, z: real  // declaration of y and z
 begin
      ...
      y = x + z // occurrences of y, x, and z
      if (...) call B // occurrence of B
 end
 Procedure C     // declaration of C
    x: real
 begin
      ...
      call B // occurrence of B
 end
begin
 ...
 call C     // occurrence of C
 call B     // occurrence of B
end
```

```
Program
 (1,1), (1,2): integer   // declarations of x and y
 Procedure (1,3)    // declaration of B
    (2,1), (2,2): real  // declaration of y and z
 begin
      ...
      (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
      if (...) call (1,3) // occurrence of B
  end
 Procedure (1,4)    // declaration of C
    (2,1): real
 begin
      ...
      call (1,3) // occurrence of B
 end
begin
 ...
 call (1,4)     // occurrence of C
 call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer // declarations of x and y
  Procedure B    // declaration of B
     y, z: real  // declaration of y and z
  begin
       ...
       y = x + z // occurrences of y, x, and z
       if (...) call B // occurrence of B
  end
  Procedure C    // declaration of C
     x: real
  begin
       ...
       call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
       ...
       (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
       if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
       ...
       call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B    // declaration of B
      y, z: real  // declaration of y and z
  begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
      x: real
  begin
        ...
        call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)  // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
        ...
        (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
        if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
        ...
        call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
      ...
      y = x + z // occurrences of y, x, and z
      if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
      ...
      call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
      ...
      (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
      if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
      ...
      call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
    y, z: real  // declaration of y and z
  begin
      ...
      y = x + z // occurrences of y, x, and z
      if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
    x: real
  begin
      ...
      call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
    (2,1), (2,2): real  // declaration of y and z
  begin
      ...
      (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
      if (...) call (1,3) // occurrence of B
  end
  Procedure (1,4)    // declaration of C
    (2,1): real
  begin
      ...
      call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
       ...
       y = x + z // occurrences of y, x, and z
       if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
       ...
       call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
       ...
       (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
       if (...) call (1,3) // occurrence of B
  end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
       ...
       call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B    // declaration of B
     y, z: real  // declaration of y and z
  begin
       ...
       y = x + z // occurrences of y, x, and z
       if (...) call B // occurrence of B
  end
  Procedure C    // declaration of C
     x: real
  begin
       ...
       call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
       ...
       (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
       if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
       ...
       call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
       ...
       y = x + z // occurrences of y, x, and z
       if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
       ...
       call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
       ...
       (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
       if (...) call (1,3) // occurrence of B
  end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
       ...
       call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
       ...
       y = x + z // occurrences of y, x, and z
       if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
       ...
       call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

➡

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
       ...
       (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
       if (...) call (1,3) // occurrence of B
  end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
       ...
       call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
        ...
        call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
        ...
        (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
        if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
        ...
        call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
    y, z: real  // declaration of y and z
  begin
      ...
      y = x + z // occurrences of y, x, and z
      if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
    x: real
  begin
      ...
      call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)   // declaration of B
    (2,1), (2,2): real  // declaration of y and z
  begin
      ...
      (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
      if (...) call (1,3) // occurrence of B
  end
  Procedure (1,4)   // declaration of C
    (2,1): real
  begin
      ...
      call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)    // occurrence of C
  call (1,3)    // occurrence of B
end
```

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for this statement:

$$(2,1) = (1,1) + (2,2)$$

What do we know?

- Assume the nesting level of the statement is **level 2**
- Register r_0 contains the current FP (frame pointer)
- **(2, 1) and (2, 2) are local variables**, so they are allocated in the activation record that current FP points to.
  (1, 1) is an non-local variable.
- Two new instructions:

  LOAD R_x, R_y means $R_x \leftarrow \text{MEM}(R_y)$

  STORE R_x, R_y means $\text{MEM}(R_x) \leftarrow R_y$

Lexical Scoping

Symbol table generated at compile time matches declarations and occurrences.
⇒ Each name can be represented as a pair (nesting_level, local_index).

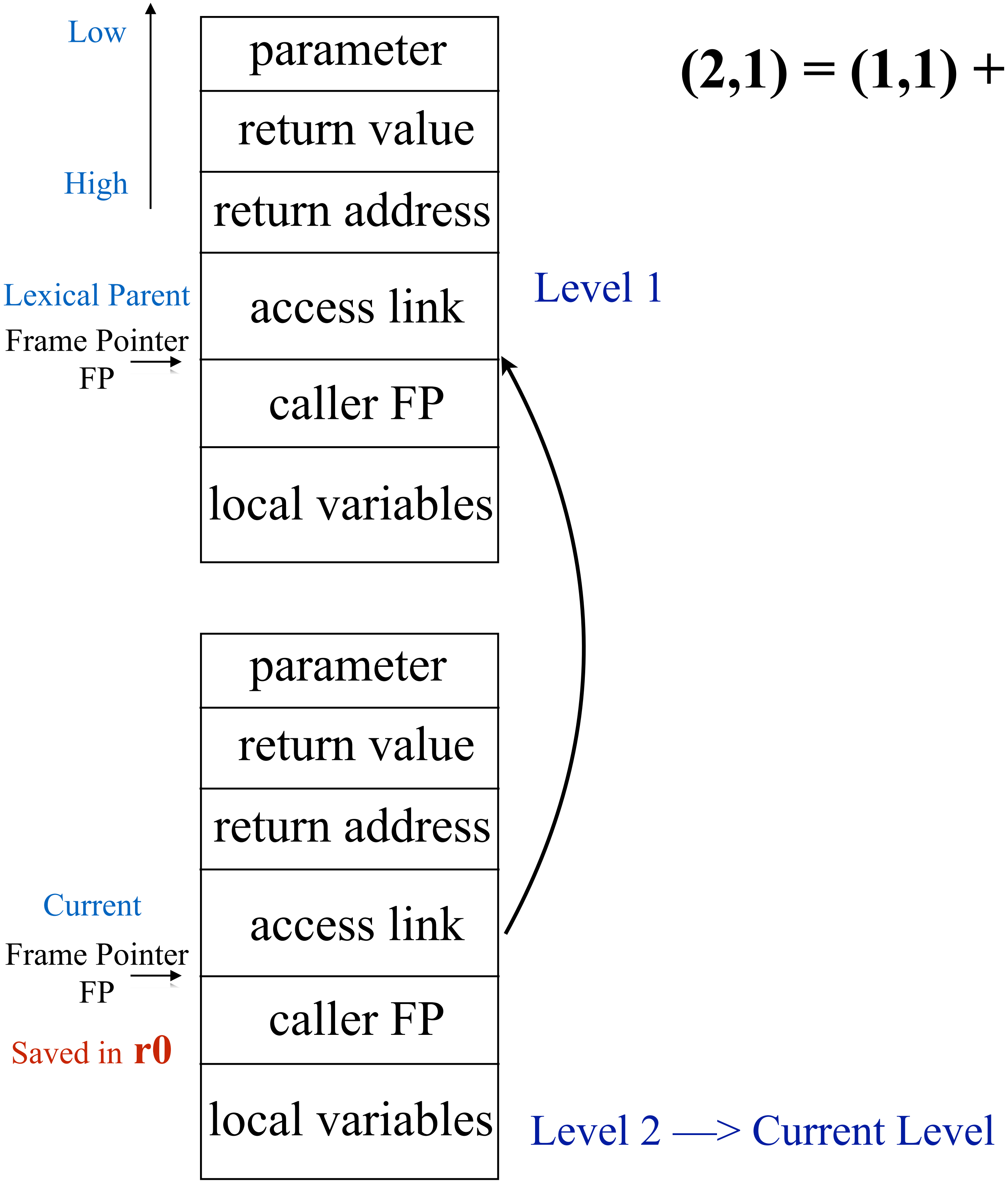
```
Program
  x, y: integer   // declarations of x and y
  Procedure B     // declaration of B
     y, z: real  // declaration of y and z
  begin
        ...
        y = x + z // occurrences of y, x, and z
        if (...) call B // occurrence of B
  end
  Procedure C     // declaration of C
     x: real
  begin
        ...
        call B // occurrence of B
  end
begin
  ...
  call C     // occurrence of C
  call B     // occurrence of B
end
```

```
Program
  (1,1), (1,2): integer   // declarations of x and y
  Procedure (1,3)    // declaration of B
     (2,1), (2,2): real  // declaration of y and z
  begin
        ...
        (2,1) = (1,1) + (2,2) // occurrences of y, x, and z
        if (...) call (1,3) // occurrence of B
   end
  Procedure (1,4)    // declaration of C
     (2,1): real
  begin
        ...
        call (1,3) // occurrence of B
  end
begin
  ...
  call (1,4)     // occurrence of C
  call (1,3)     // occurrence of B
end
```

Review: Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?

(2,1) = (1,1) + (2,2)



Low

High

Lexical Parent

Frame Pointer

FP

parameter

return value

return address

access link

caller FP

local variables

Level 1

Current

Frame Pointer

FP

Saved in **r0**

parameter

return value

return address

access link

caller FP

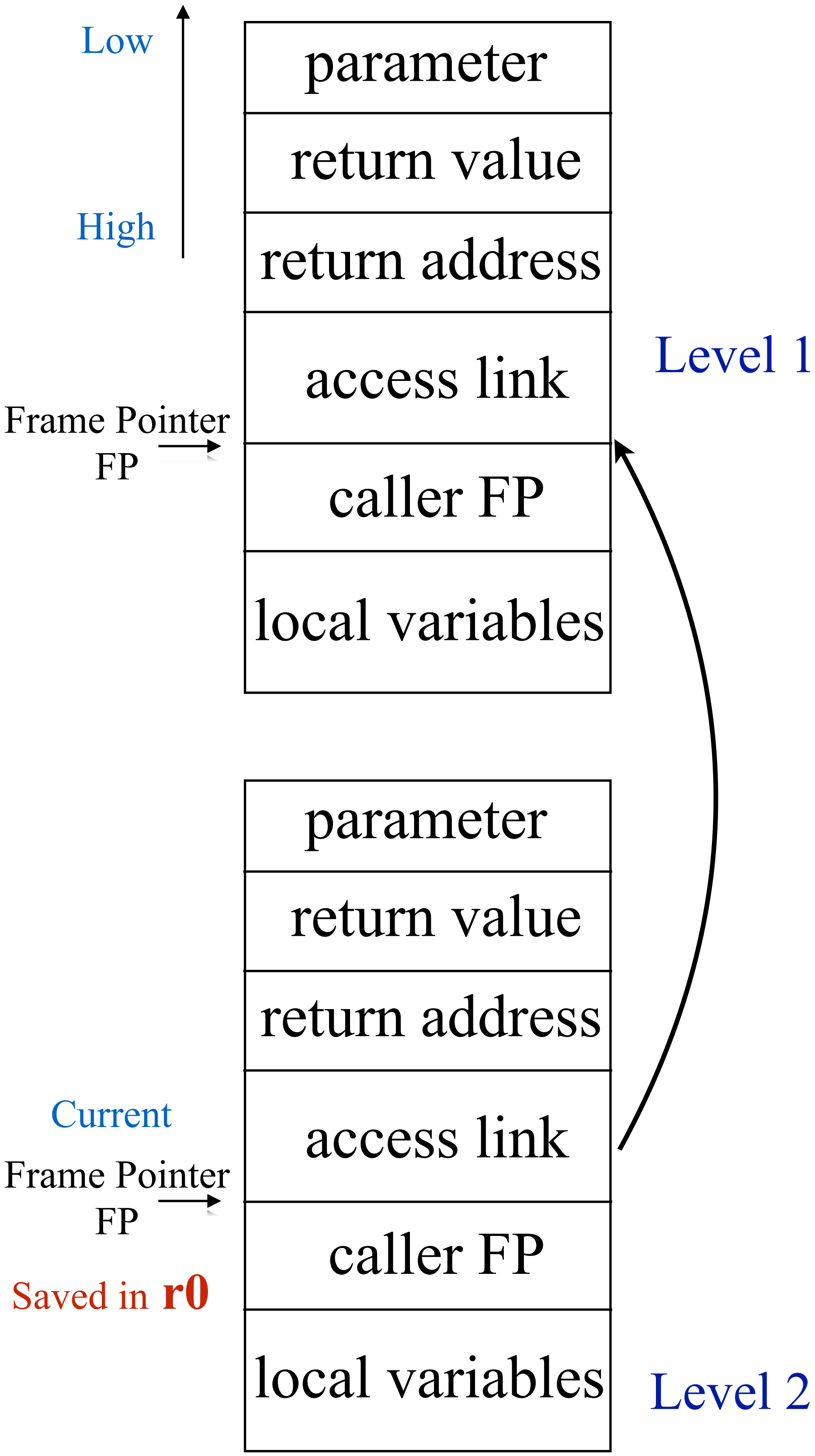
local variables

Level 2 —> Current Level

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



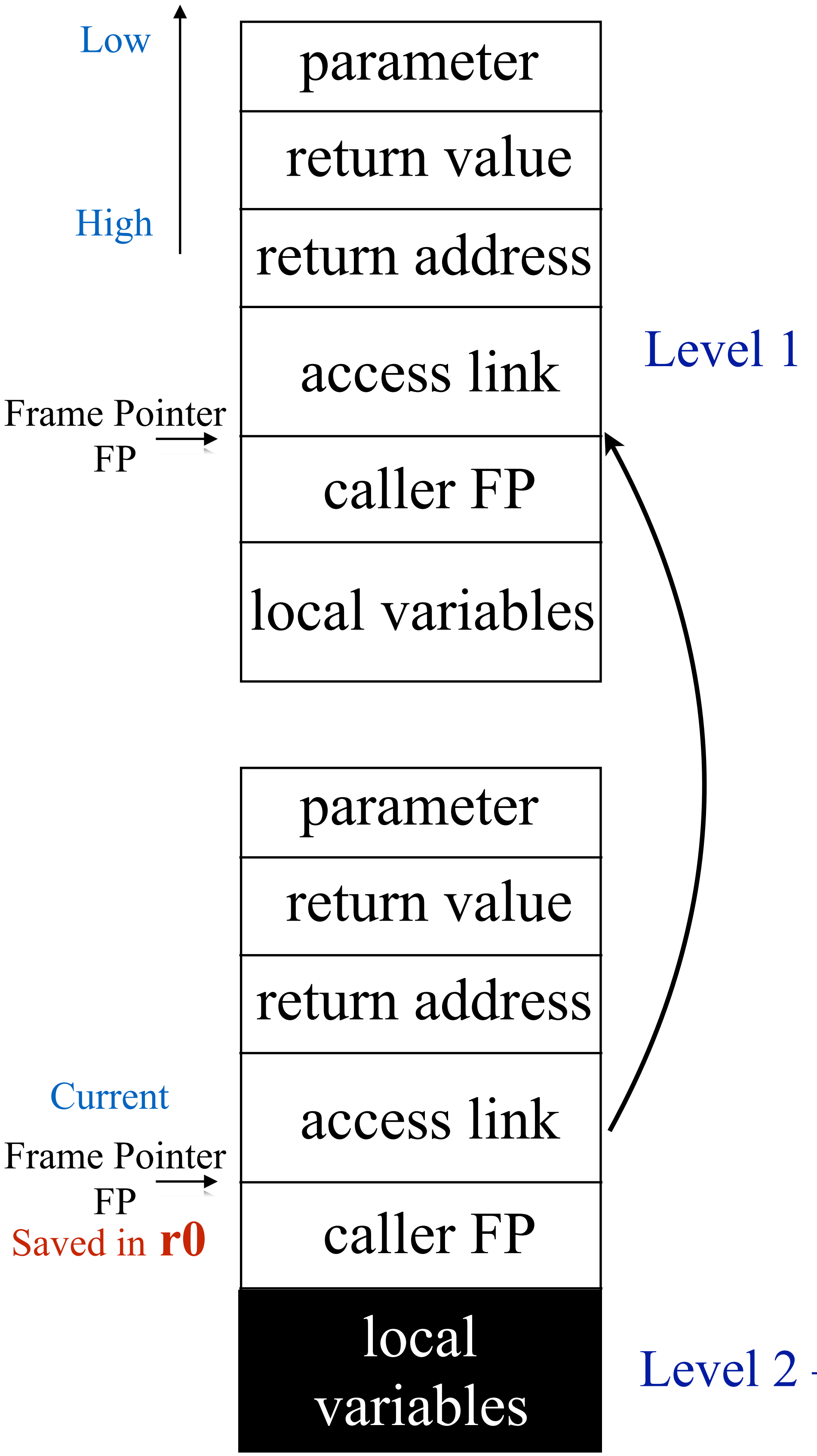
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4      // offset of local variable (1,1) in frame
       LOADI r2, #-4     // offset of access link in frame (bytes)
       ADD r3 r0 r2      // address of access link
       LOAD r4 r3        // get access link
       ADD r5 r4 r1      // address of local variable (1,1) in frame
       LOAD r6 r5        // get content of variable (1,1)
(2,2)  LOADI r7 #8       // offset of local variable (2,2) in frame
       ADD r8 r0 r7      // address of local variable (2,2)
       LOAD r9 r8        // get content of variable (2, 2)
  +    ADD r10 r6 r9     // (1,1) + (2,2)
(2,1)  LOADI r11 #4      // offset of local variable (2,1) in frame
       ADD r12 r0 r11    // address of local variable (2,1)
  =    STORE r12 r10     // (2,1) = (1,1) + (2,2)
```

Assume "access link" field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



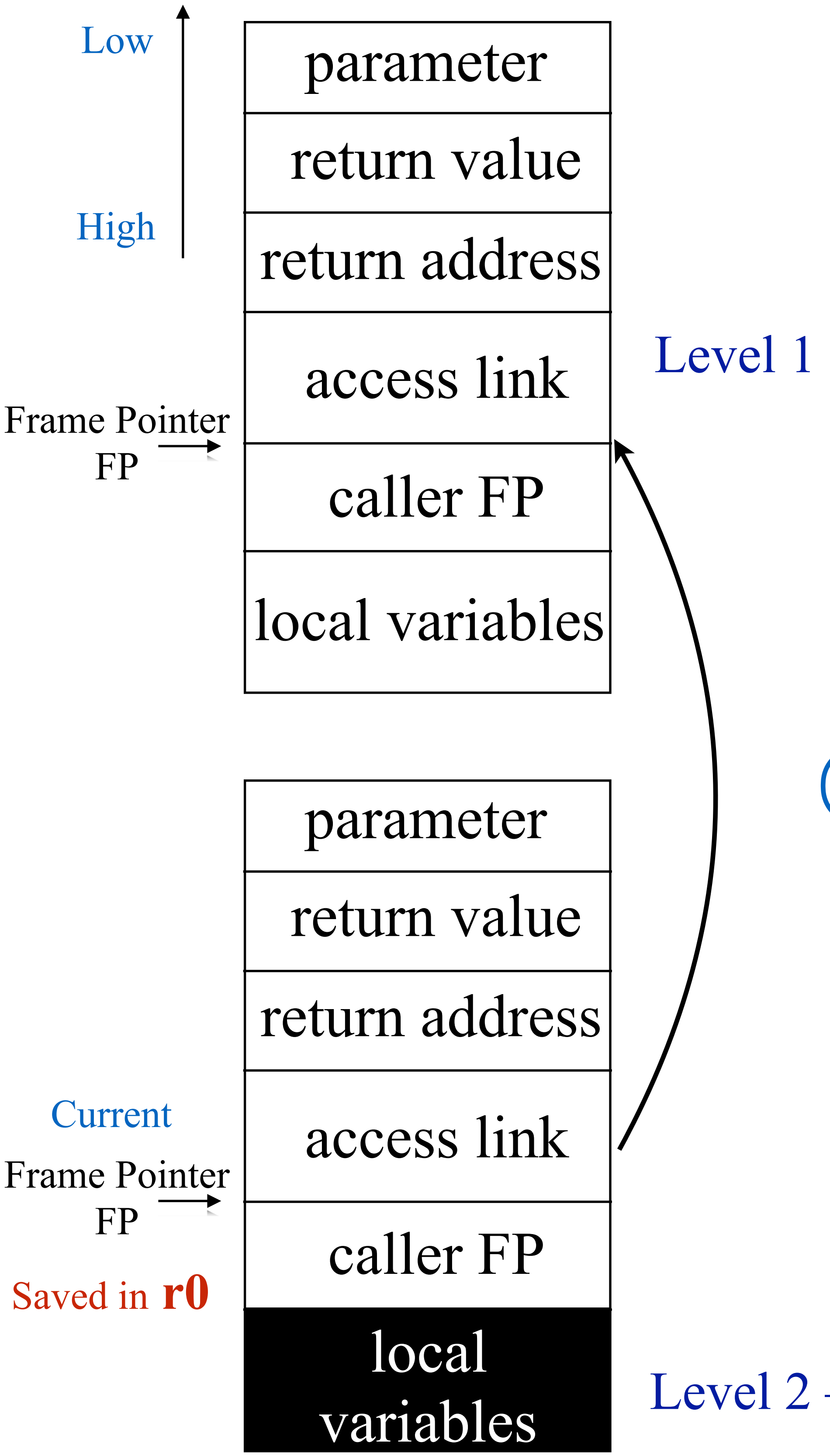
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4     // offset of local variable (1,1) in frame
       LOADI r2, #-4    // offset of access link in frame (bytes)
       ADD r3 r0 r2     // address of access link
       LOAD r4 r3       // get access link
       ADD r5 r4 r1     // address of local variable (1,1) in frame
       LOAD r6 r5       // get content of variable (1,1)
(2,2)  LOADI r7 #8      // offset of local variable (2,2) in frame
       ADD r8 r0 r7     // address of local variable (2,2)
       LOAD r9 r8       // get content of variable (2, 2)
  +    ADD r10 r6 r9    // (1,1) + (2,2)
(2,1)  LOADI r11 #4     // offset of local variable (2,1) in frame
       ADD r12 r0 r11   // address of local variable (2,1)
  =    STORE r12 r10    // (2,1) = (1,1) + (2,2)
```

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



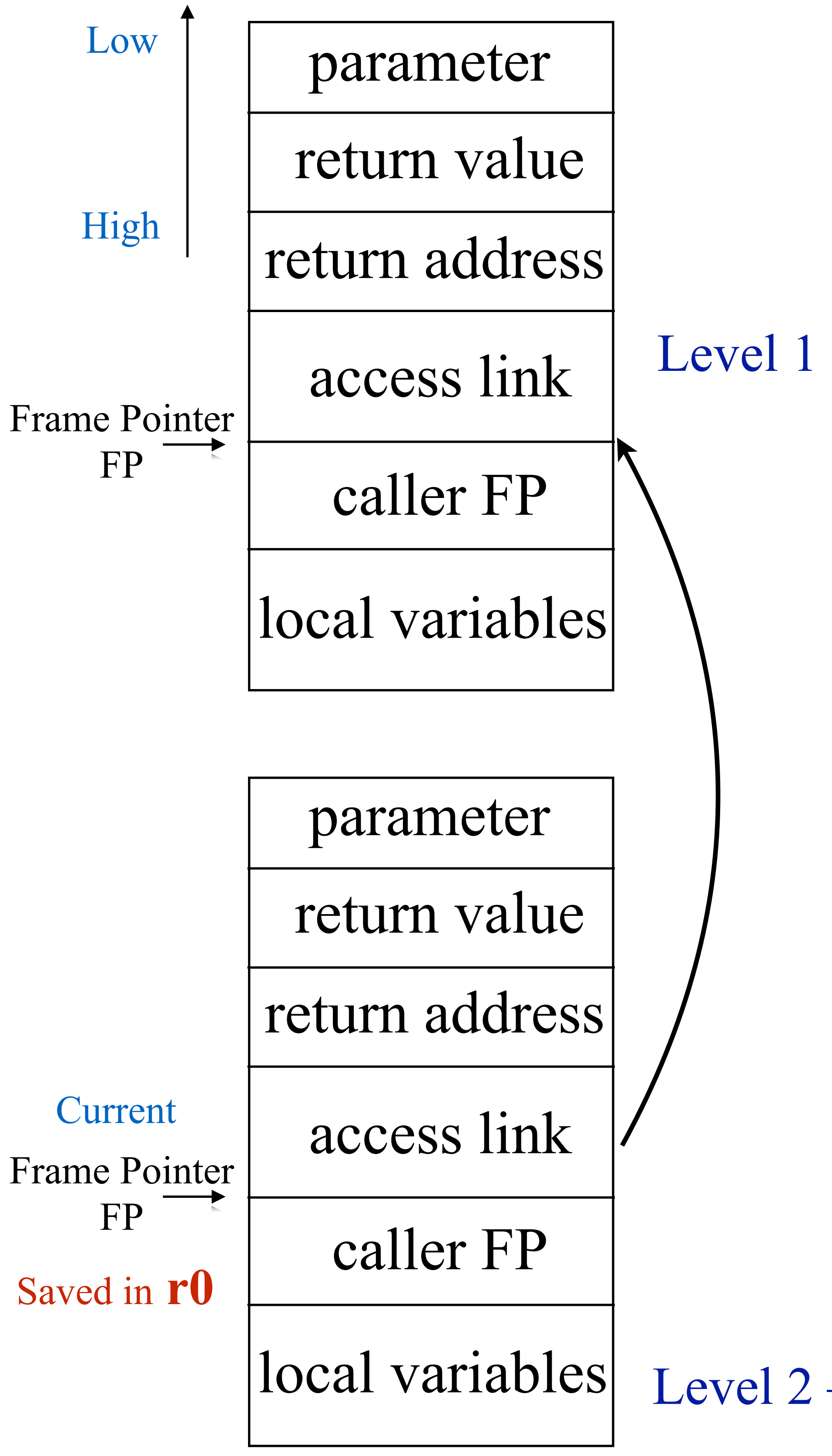
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4      // offset of local variable (1,1) in frame
       LOADI r2, #-4     // offset of access link in frame (bytes)
       ADD r3 r0 r2      // address of access link
       LOAD r4 r3        // get access link
       ADD r5 r4 r1      // address of local variable (1,1) in frame
       LOAD r6 r5        // get content of variable (1,1)
(2,2)  LOADI r7 #8       // offset of local variable (2,2) in frame
       ADD r8 r0 r7      // address of local variable (2,2)
       LOAD r9 r8        // get content of variable (2, 2)
  +    ADD r10 r6 r9     // (1,1) + (2,2)
(2,1)  LOADI r11 #4      // offset of local variable (2,1) in frame
       ADD r12 r0 r11    // address of local variable (2,1)
  =    STORE r12 r10     // (2,1) = (1,1) + (2,2)
```

Assume "access link" field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



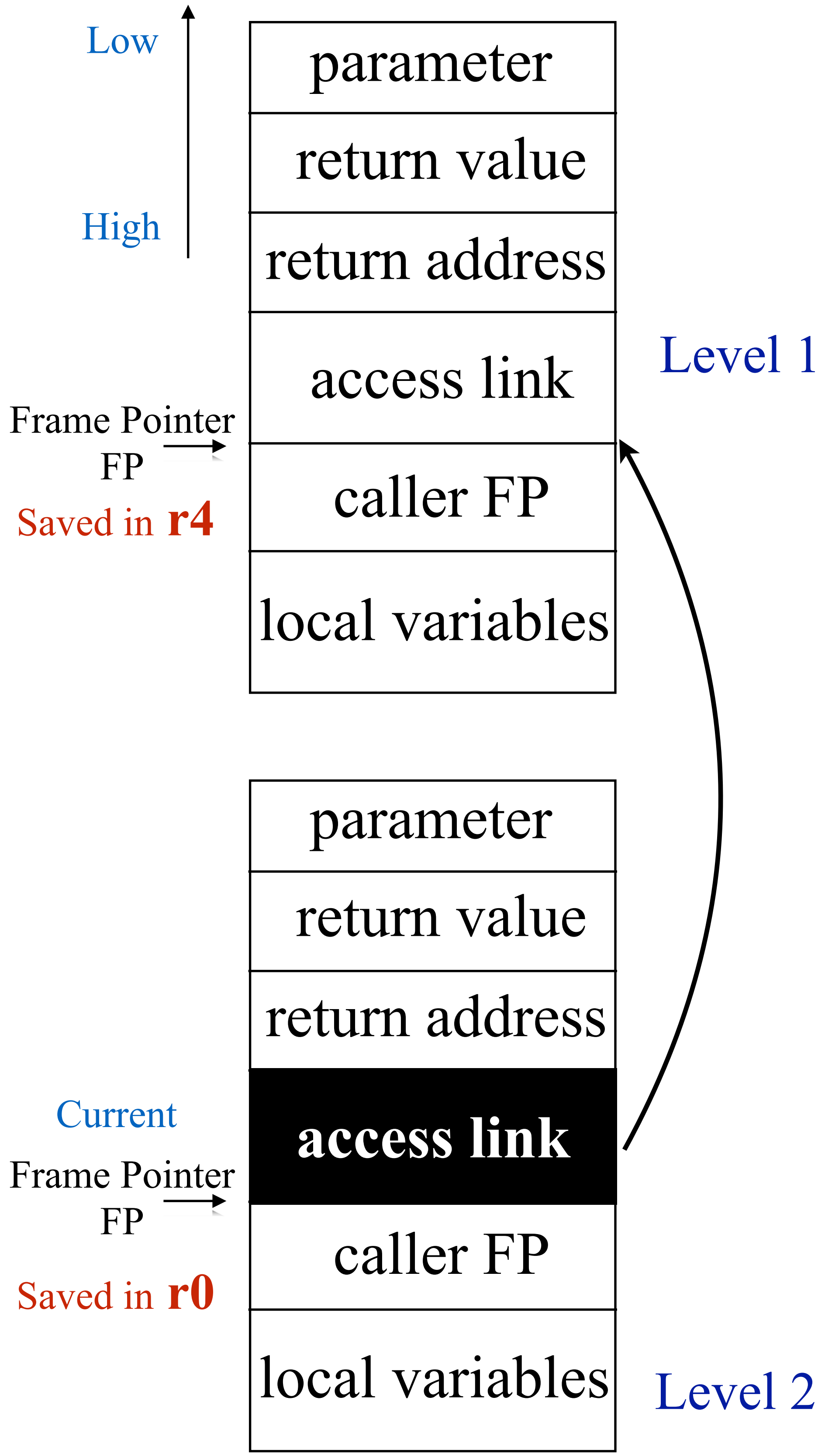
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4      // offset of local variable (1,1) in frame
       LOADI r2, #-4     // offset of access link in frame (bytes)
       ADD r3 r0 r2      // address of access link
       LOAD r4 r3        // get access link
       ADD r5 r4 r1      // address of local variable (1,1) in frame
       LOAD r6 r5        // get content of variable (1,1)
(2,2)  LOADI r7 #8       // offset of local variable (2,2) in frame
       ADD r8 r0 r7      // address of local variable (2,2)
       LOAD r9 r8        // get content of variable (2, 2)
  +    ADD r10 r6 r9     // (1,1) + (2,2)
(2,1)  LOADI r11 #4      // offset of local variable (2,1) in frame
       ADD r12 r0 r11    // address of local variable (2,1)
  =    STORE r12 r10     // (2,1) = (1,1) + (2,2)
```

Assume "access link" field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



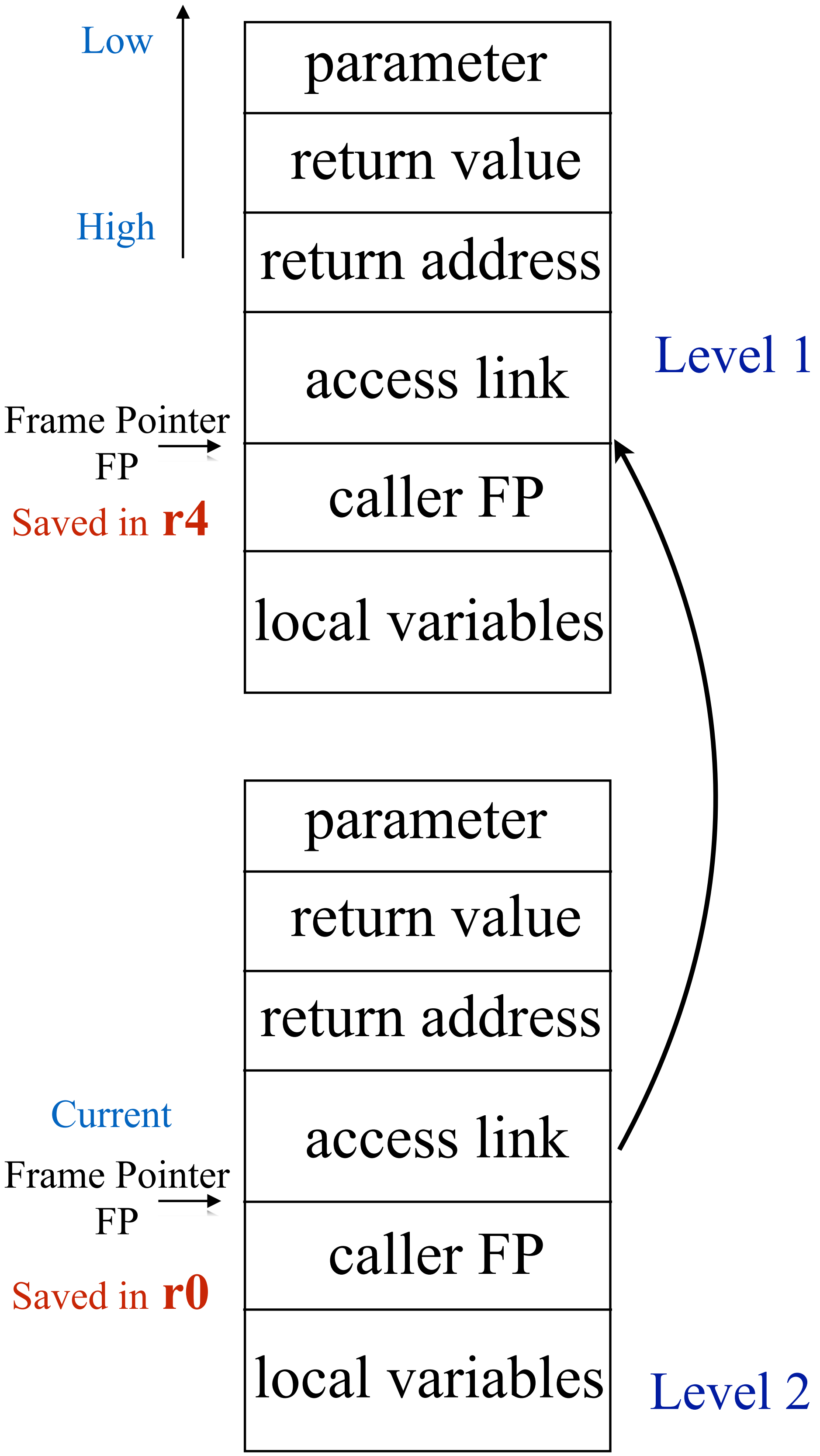
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4      // offset of local variable (1,1) in frame
       LOADI r2, #-4     // offset of access link in frame (bytes)
       ADD r3 r0 r2      // address of access link
       LOAD r4 r3        // get access link, fp for frame at level 1
       ADD r5 r4 r1      // address of local variable (1,1) in frame
       LOAD r6 r5        // get content of variable (1,1)
(2,2)  LOADI r7 #8       // offset of local variable (2,2) in frame
       ADD r8 r0 r7      // address of local variable (2,2)
       LOAD r9 r8        // get content of variable (2, 2)
  +    ADD r10 r6 r9     // (1,1) + (2,2)
(2,1)  LOADI r11 #4      // offset of local variable (2,1) in frame
       ADD r12 r0 r11    // address of local variable (2,1)
  =    STORE r12 r10     // (2,1) = (1,1) + (2,2)
```

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



Low

High

parameter

return value

return address

access link

caller FP

local variables

Level 1

Frame Pointer FP

Saved in **r4**

parameter

return value

return address

access link

caller FP

local variables

Current Frame Pointer FP

Saved in **r0**

Level 2 —> Current Level

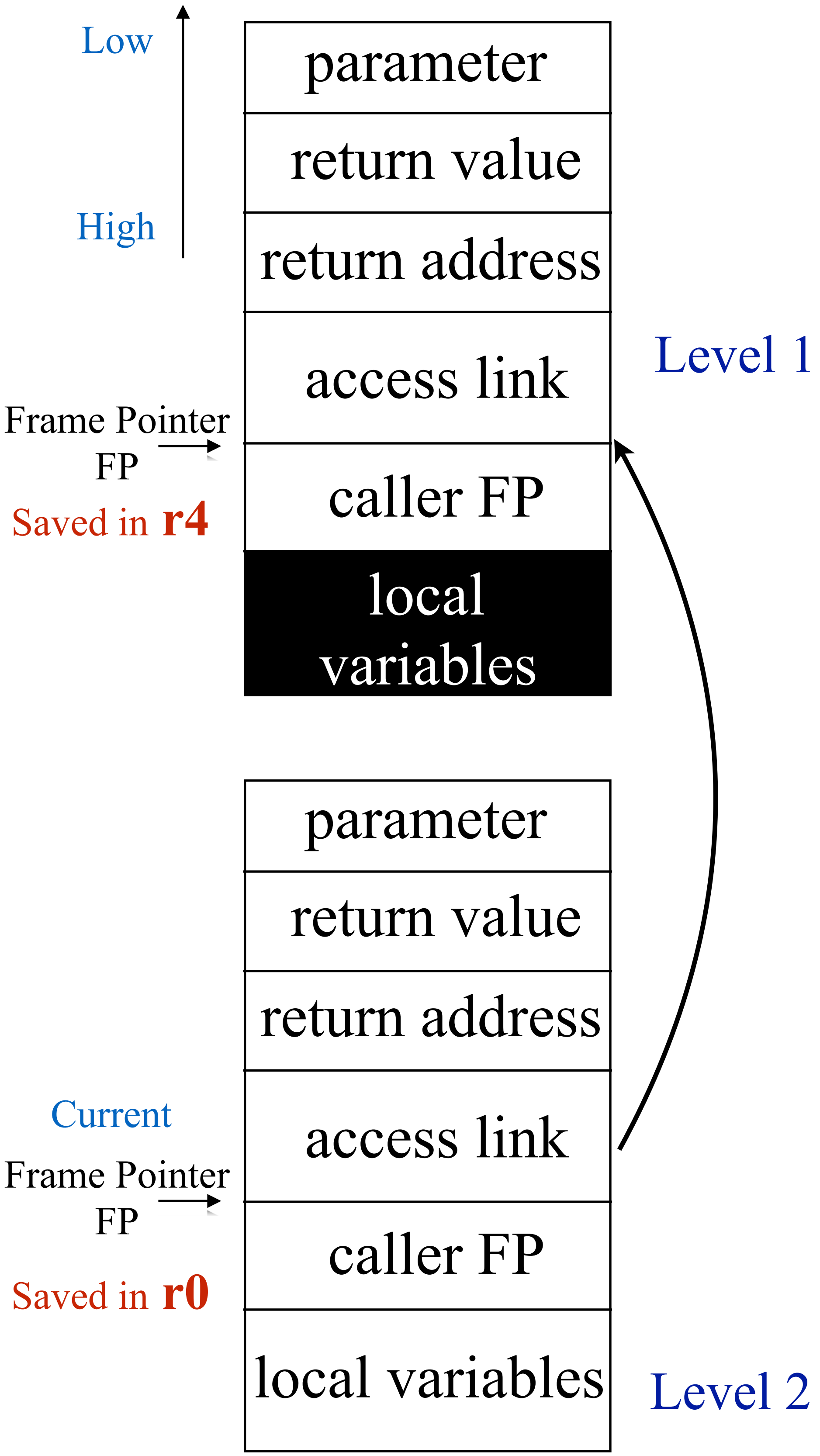
(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4      // offset of local variable (1,1) in frame
       LOADI r2, #-4     // offset of access link in frame (bytes)
       ADD r3 r0 r2      // address of access link
       LOAD r4 r3        // get access link, fp for frame at level 1
       ADD r5 r4 r1      // address of local variable (1,1) in frame
       LOAD r6 r5        // get content of variable (1,1)
(2,2)  LOADI r7 #8       // offset of local variable (2,2) in frame
       ADD r8 r0 r7      // address of local variable (2,2)
       LOAD r9 r8        // get content of variable (2, 2)
  +    ADD r10 r6 r9     // (1,1) + (2,2)
(2,1)  LOADI r11 #4      // offset of local variable (2,1) in frame
       ADD r12 r0 r11    // address of local variable (2,1)
  =     STORE r12 r10    // (2,1) = (1,1) + (2,2)
```

Assume "access link" field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Access to Non-Local Data(Lexical Scoping)

What code do we need to generate for statement (*)?



Low

High

parameter

return value

return address

access link

Frame Pointer FP

Saved in **r4**

caller FP

local variables

Level 1

parameter

return value

return address

Current Frame Pointer FP

access link

Saved in **r0**

caller FP

local variables

(2,1) = (1,1) + (2,2)

```
(1,1)  LOADI r1, #4     // offset of local variable (1,1) in frame
       LOADI r2, #-4    // offset of access link in frame (bytes)
       ADD r3 r0 r2     // address of access link
       LOAD r4 r3       // get access link, fp for frame at level 1
       ADD r5 r4 r1     // address of local variable (1,1) in frame
       LOAD r6 r5       // get content of variable (1,1)
(2,2)  LOADI r7 #8      // offset of local variable (2,2) in frame
       ADD r8 r0 r7     // address of local variable (2,2)
       LOAD r9 r8       // get content of variable (2, 2)
  +    ADD r10 r6 r9    // (1,1) + (2,2)
(2,1)  LOADI r11 #4     // offset of local variable (2,1) in frame
       ADD r12 r0 r11   // address of local variable (2,1)
  =    STORE r12 r10    // (2,1) = (1,1) + (2,2)
```

Level 2 —> Current Level

Assume “access link” field is 4 bytes.
Each unit offset corresponds to 4 bytes.

Next Lecture

Things to do:

- Read Scott, Chapter 3.1 - 3.4, Chapter 9.1 - 9.3 (4th Edition) or Chapter 8.1 - 8.3 (3rd Edition)
- Read ALSU, Chapter 7.1 - 7.3 (2nd Edition).