What’s in a name? — Each name “means” something!

• Denotes a programming language construct

• 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 value, a memory address.
Review: Names, Bindings and Memory

**Bindings** – Association of a name with the thing it “names” (e.g., a name and a memory location, a function name and its “meaning”, a name and a value)

- **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

Compiler needs bindings to know meaning of names during translation (and execution).
Review: 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).
Review: How to Maintain Bindings

- Symbol table: maintained by compiler during compilation
- Referencing Environment: maintained by compiler-generated-code during program execution

Question:
- How long do bindings last for a name hold in a program?
- What initiates a binding?
- What ends a binding?

Scope of a binding: the part of the in which the binding is active.
Nested Subroutines (Algol 60, Ada, ML, 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, ML, 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, ML, 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, ML, Common Lisp, Python, ….)

```plaintext
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, ML, 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

Example:
- The reference to \( n \) in \( W \) is associated with two different declarations at two different times
- 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

• Example:
  - The reference to n in W is associated with two different declarations at two different times
  - 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;
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

• Example:
  - The reference to n in W is associated with two different declarations at two different times
  - 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
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

• Example:
  - The reference to n in W is associated with two different declarations at two different times
  - 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;
end
```

Calling Chain:

\[ L \Rightarrow W \]
\[ L \Rightarrow D \Rightarrow W \]
Context of Procedures

Two contexts

• *static* placement in source code (same for each invocation)
• *dynamic* run-time stack context (different for each invocation)

Scope Rules:
Each variable reference must be associated with a single declaration.

Two choices:
1. Use static and dynamic context: *lexical scope*
2. Use dynamic context: *dynamic scope*

• Easy for variables declared locally, and same for *lexical* and *dynamic* scoping
• Harder for variables not declared locally and not same for *lexical* and *dynamic* scoping
Two important steps

1. **Compile-time**: How do we map a name into a (level, offset) pair? We use a block structured symbol table (**compile time**)
   - When we look up a name, we want to get the most recent declaration for the name
   - The declaration may be found in the current procedure or in any ancestor procedure

2. **Run-time**: Given a (level, offset) pair, what’s the address?
   - Two classical approaches:
     ⇒ access link (**static link**)  
     ⇒ display
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
Two important steps

1. *Compile-time*: How do we map a name into a (level, offset) pair?
   We use a block structured symbol table (*compile time*)
   - When we look up a name, we want to get the most recent declaration for the name
   - The declaration may be found in the current procedure or in any ancestor procedure

2. *Run-time*: Given a (level, offset) pair, what’s the address?
   - Two classical approaches:
     - access link (*static link*)
     - display
Access to Non-Local Data (Lexical Scoping): Access Link

Using **access links** (static links)

- Each AR has a pointer to most recent AR of immediate lexical ancestor
- Lexical ancestor does not need to be the caller

---

**Activation Record Pointer**

- parameter
- register
- save area
- return value
- return address
- access link
- caller’s ARP
- local variables

---

** caller’s ARP**

- parameter
- register
- save area
- return value
- return address
- access link
- local variables

---

** caller’s ARP**

- parameter
- register
- save area
- return value
- return address
- access link
- local variables

---

Example: reference to <p, 16> runs up access link to p

*Cost of access link is proportional to lexical distance*
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:
  \[\text{LOAD } R_x, R_y\]  means \( R_x \leftarrow \text{MEM}(R_y) \)
  \[\text{STORE } R_x, R_y\]  means \( \text{MEM}(R_x) \leftarrow R_y \)
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

Direction of stack growth (usually toward lower addresses)

Direction of stack growth (usually toward lower addresses)

<table>
<thead>
<tr>
<th>parameter</th>
<th>(1,1) LOADI r1, #4 // offset of local variable (1,1) in frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>return value</td>
<td>LOADI r2, #-4 // offset of access link in frame (bytes)</td>
</tr>
<tr>
<td>return address</td>
<td>ADD r3 r0 r2 // address of access link</td>
</tr>
<tr>
<td>access link</td>
<td>LOAD r4 r3 // get access link</td>
</tr>
<tr>
<td>caller FP</td>
<td>ADD r5 r4 r1 // address of local variable (1,1) in frame</td>
</tr>
<tr>
<td>local variables</td>
<td>LOAD r6 r5 // get content of variable (1,1)</td>
</tr>
</tbody>
</table>
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

- \(\text{LOADI } r1, \#4\) // offset of local variable \((1,1)\) in frame
- \(\text{LOADI } r2, \#-4\) // offset of access link in frame (bytes)
- \(\text{ADD } r3 r0 r2\) // address of access link
- \(\text{LOAD } r4 r3\) // get access link
- \(\text{ADD } r5 r4 r1\) // address of local variable \((1,1)\) in frame
- \(\text{LOAD } r6 r5\) // get content of variable \((1,1)\)
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)
```

Direction of stack growth (usually lower addresses)

Frame Pointer (FP)

- Parameter
- Return value
- Return address
- Access link
- Caller FP
- Local variables
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)
```
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
LOAD 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)
```
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

<table>
<thead>
<tr>
<th>Direction of stack growth (usually lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer FP</td>
</tr>
<tr>
<td>--&gt;</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

\[(2,1) = (1,1) + (2,2)\]

<table>
<thead>
<tr>
<th>Direction of stack growth (usually lower addresses)</th>
<th>parameter</th>
<th>(1,1)</th>
<th>LOADI r1, #4 // offset of local variable (1,1) in frame</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>return value</td>
<td>LOADI r2, #-4 // offset of access link in frame (bytes)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>return address</td>
<td>ADD r3 r0 r2 // address of access link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>access link</td>
<td>LOAD r4 r3 // get access link</td>
<td></td>
</tr>
<tr>
<td></td>
<td>caller FP</td>
<td>ADD r5 r4 r1 // address of local variable (1,1) in frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>local variables</td>
<td>LOAD r6 r5 // get content of variable (1,1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(2,2)</td>
<td>LOADI r7 #8 // offset of local variable (2,2) in frame</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADD r8 r0 r7 // address of local variable (2,2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LOAD r9 r8 // get content of variable (2,2)</td>
<td></td>
</tr>
</tbody>
</table>
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

<table>
<thead>
<tr>
<th>Direction of stack growth (usually lower addresses)</th>
<th>parameter</th>
<th>return value</th>
<th>return address</th>
<th>access link</th>
<th>caller FP</th>
<th>local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer FP</td>
<td>(1,1) LOADI r1, #4 \quad // offset of local variable (1,1) in frame</td>
<td>LOADI r2, #-4 \quad // offset of access link in frame (bytes)</td>
<td>ADD r3 r0 r2 \quad // address of access link</td>
<td>LOAD r4 r3 \quad // get access link</td>
<td>ADD r5 r4 r1 \quad // address of local variable (1,1) in frame</td>
<td>LOAD r6 r5 \quad // get content of variable (1,1)</td>
</tr>
<tr>
<td></td>
<td>(2,2) LOADI r7 #8 \quad // offset of local variable (2,2) in frame</td>
<td>ADD r8 r0 r7 \quad // address of local variable (2,2)</td>
<td>LOAD r9 r8 \quad // get content of variable (2,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Access to Non-Local Data (Lexical Scoping)

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

\[(2, 1) = (1, 1) + (2, 2)\]

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

\[(2,1) = (1,1) + (2,2)\]

<table>
<thead>
<tr>
<th>Direction of stack growth (usually lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame Pointer FP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1) LOADI r1, #4  // offset of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOADI r2, #-4  // offset of access link in frame (bytes)</td>
</tr>
<tr>
<td>ADD r3 r0 r2  // address of access link</td>
</tr>
<tr>
<td>LOAD r4 r3  // get access link</td>
</tr>
<tr>
<td>ADD r5 r4 r1  // address of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOAD r6 r5  // get content of variable (1,1)</td>
</tr>
<tr>
<td>LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,2) LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,2) LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Access link</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,1) LOADI r1, #4  // offset of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOADI r2, #-4  // offset of access link in frame (bytes)</td>
</tr>
<tr>
<td>ADD r3 r0 r2  // address of access link</td>
</tr>
<tr>
<td>LOAD r4 r3  // get access link</td>
</tr>
<tr>
<td>ADD r5 r4 r1  // address of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOAD r6 r5  // get content of variable (1,1)</td>
</tr>
<tr>
<td>LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Caller FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2,2) LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1) LOADI r1, #4  // offset of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOADI r2, #-4  // offset of access link in frame (bytes)</td>
</tr>
<tr>
<td>ADD r3 r0 r2  // address of access link</td>
</tr>
<tr>
<td>LOAD r4 r3  // get access link</td>
</tr>
<tr>
<td>ADD r5 r4 r1  // address of local variable (1,1) in frame</td>
</tr>
<tr>
<td>LOAD r6 r5  // get content of variable (1,1)</td>
</tr>
<tr>
<td>LOADI r7 #8  // offset of local variable (2,2) in frame</td>
</tr>
<tr>
<td>ADD r8 r0 r7  // address of local variable (2,2)</td>
</tr>
<tr>
<td>LOAD r9 r8  // get content of variable (2,2)</td>
</tr>
</tbody>
</table>

\[+ ADD r10 r6 r9  // (1,1) + (2,2)\]
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]
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)
```
Access to Non-Local Data (Lexical Scoping)

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

\[(2,1) = (1,1) + (2,2)\]

**Code:**

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

- Read ALSU, Chapter 7.1 - 7.3 (2nd Edition).