CS 403: Subprograms

Stefan D. Bruda

Fall 2021

Procedures and Parameters

- Two fundamental abstraction facilities: data abstraction and process abstraction
- In particular subprograms simplify complex programs though abstraction
  - Abstraction of actions
  - Called by name with arguments: `Calculate_Pay(pay_rate, hours_worked)`
  - Single entry (caller is suspended for the duration)
  - Control always returns to the caller when the subprogram terminates
  - Procedures (subroutines) and functions (or methods in OOP languages)
- Design issues for subprograms
  - Are local variables static or dynamic?
    - The local reference environment may be static (historical significance only)
    - The local reference environment may be stack-based (all modern languages)
  - What are the parameter passing methods?
  - Are the types of the actual and formal parameters checked?
  - Can subprograms be passed as parameters? What is the referencing environment?
  - Can subprogram definitions be nested?
  - Can subprograms be overloaded or generic?
  - Are side effects allowed?
  - What type of variables can be returned?

Parameter Passing Mechanisms

- Pass by value
  - Ada: The arguments are expressions evaluated at the time of the call
    - Parameters are constant values
    - All the parameters in the body of the procedure will be replaced by those values

Parameter Passing Mechanisms

- Pass by value
  - Ada: The arguments are expressions evaluated at the time of the call
    - Parameters are constant values
    - All the parameters in the body of the procedure will be replaced by those values
  - Pascal, C: arguments are still expressions evaluated at the time of the call
    - Now the parameters are local variables, initialized by the arguments from the call
    - The main method in most programming languages

Parameter Passing Mechanisms

- Pass by value
  - Ada: The arguments are expressions evaluated at the time of the call
    - Parameters are constant values
    - All the parameters in the body of the procedure will be replaced by those values
  - Pascal, C: arguments are still expressions evaluated at the time of the call
    - Now the parameters are local variables, initialized by the arguments from the call
    - The main method in most programming languages
- Pass by reference
  - The arguments must be variables; then the location of the variable is passed so the parameter becomes an alias for the argument
  - Examples of use:
    - `var` prefix in Pascal and Modula-2
    - A reference passed explicitly (`&` and `*`) in C and Algol
    - Arrays are always passed by reference in C and Ada-95
    - Objects are always passed by reference in Java

Parameter Passing Mechanisms

- Pass by value
  - Ada: The arguments are expressions evaluated at the time of the call
    - Parameters are constant values
    - All the parameters in the body of the procedure will be replaced by those values
  - Pascal, C: arguments are still expressions evaluated at the time of the call
    - Now the parameters are local variables, initialized by the arguments from the call
    - The main method in most programming languages
- Pass by reference
  - The arguments must be variables; then the location of the variable is passed so the parameter becomes an alias for the argument
  - Examples of use:
    - `var` prefix in Pascal and Modula-2
    - A reference passed explicitly (`&` and `*`) in C and Algol
    - Arrays are always passed by reference in C and Ada-95
    - Objects are always passed by reference in Java
- Pass by name/lazy evaluation
  - The textual representation of the argument replaces the name of the parameter throughout the body of the function, or
  - Like pass by value but the argument is not evaluated until its first actual use
  - Examples of use: Algol60 and many functional languages

Type Checking of Parameters

- Strongly typed languages require parameters to be checked in type and number
- Procedures cannot have a variable number of parameters
- Pass by reference: parameters and arguments must have the same type
- Pass by value: the condition above is relaxed to assignment compatibility

Subprograms as Parameters

- Subprogram parameters still need to be type checked
- The referencing environment can be:
  - Shallow binding $\rightarrow$ the environment of the call statement that enacts the passed subprogram
  - Deep binding $\rightarrow$ the environment of the definition of the passed subprogram (lexical closure)
  - Ad-hoc binding $\rightarrow$ the environment of the call statement that passed the subprogram
- Example: execution of `SUB2` when called by `SUB4`

```
procedure SUB1;
    (* The static parent of
       the passed program *)
    var x:  integer;
    procedure SUB2;
        begin
            write('x = ', x)
        end;
    procedure SUB3;
        var x:  integer;
        begin
            x := 3;
            SUB4(SUB2)
            (* the call stmt
               that "enacts"
               SUB2 *)
        end;
    procedure SUB4(SUBX);
        var x:  integer;
        begin
            x := 4;
            SUBX
        end;
    begin
        x := 1;
        SUB3
    end;
```

Subprograms as Parameters

- Subprogram parameters still need to be type checked
- The referencing environment can be:
  - Shallow binding $\rightarrow$ the environment of the call statement that enacts the passed subprogram
  - Deep binding $\rightarrow$ the environment of the definition of the passed subprogram (lexical closure)
  - Ad-hoc binding $\rightarrow$ the environment of the call statement that passed the subprogram
- Example: execution of `SUB2` when called by `SUB4`
  - Shallow binding: `x = 4` (the referencing environment is that of `SUB4`)

```
procedure SUB1;
    (* The static parent of
       the passed program *)
    var x:  integer;
    procedure SUB2;
        begin
            write('x = ', x)
        end;
    procedure SUB3;
        var x:  integer;
        begin
            x := 3;
            SUB4(SUB2)
            (* the call stmt
               that "enacts"
               SUB2 *)
        end;
    procedure SUB4(SUBX);
        var x:  integer;
        begin
            x := 4;
            SUBX
        end;
    begin
        x := 1;
        SUB3
    end;
```

Subprograms as Parameters

- Subprogram parameters still need to be type checked
- The referencing environment can be:
  - Shallow binding $\rightarrow$ the environment of the call statement that enacts the passed subprogram
  - Deep binding $\rightarrow$ the environment of the definition of the passed subprogram (lexical closure)
  - Ad-hoc binding $\rightarrow$ the environment of the call statement that passed the subprogram
- Example: execution of `SUB2` when called by `SUB4`
  - Shallow binding: `x = 4` (the referencing environment is that of `SUB4`)
  - Deep binding: `x = 1` (the referencing environment is that of `SUB1`, the static parent of `SUB2`)

```
procedure SUB1;
    (* The static parent of
       the passed program *)
    var x:  integer;
    procedure SUB2;
        begin
            write('x = ', x)
        end;
    procedure SUB3;
        var x:  integer;
        begin
            x := 3;
            SUB4(SUB2)
            (* the call stmt
               that "enacts"
               SUB2 *)
        end;
    procedure SUB4(SUBX);
        var x:  integer;
        begin
            x := 4;
            SUBX
        end;
    begin
        x := 1;
        SUB3
    end;
```

Subprograms as Parameters

- Subprogram parameters still need to be type checked
- The referencing environment can be:
  - Shallow binding $\rightarrow$ the environment of the call statement that enacts the passed subprogram
  - Deep binding $\rightarrow$ the environment of the definition of the passed subprogram (lexical closure)
  - Ad-hoc binding $\rightarrow$ the environment of the call statement that passed the subprogram
- Example: execution of `SUB2` when called by `SUB4`
  - Shallow binding: `x = 4` (the referencing environment is that of `SUB4`)
  - Deep binding: `x = 1` (the referencing environment is that of `SUB1`, the static parent of `SUB2`)
  - Ad-hoc binding: `x = 3` (the referencing environment is that of `SUB3`)

```
procedure SUB1;
    (* The static parent of
       the passed program *)
    var x:  integer;
    procedure SUB2;
        begin
            write('x = ', x)
        end;
    procedure SUB3;
        var x:  integer;
        begin
            x := 3;
            SUB4(SUB2)
            (* the call stmt
               that "enacts"
               SUB2 *)
        end;
    procedure SUB4(SUBX);
        var x:  integer;
        begin
            x := 4;
            SUBX
        end;
    begin
        x := 1;
        SUB3
    end;
```

Deep Binding in Haskell

```
increment :: Int -> Int
increment = (+) x  -- the first argument of (+) is bound to its
    where x = 1    -- value at the point of the definition of
                   -- increment

foo :: Int -> Int
foo x = 2 * increment x
```

```
Main> foo 10
22                 -- it would be 40 with shallow binding
```

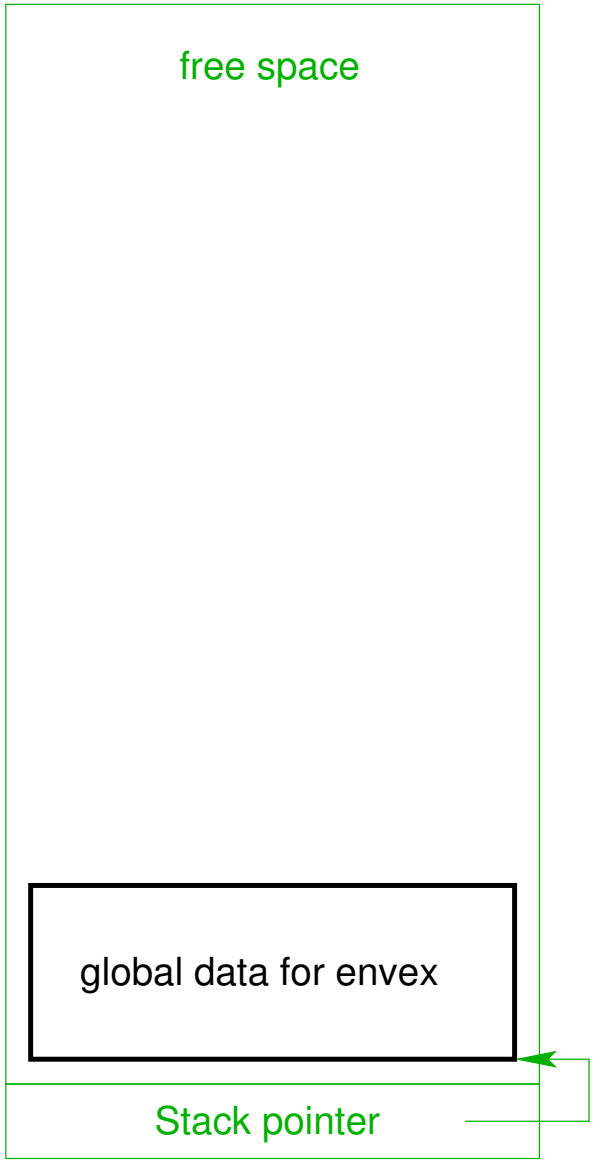
Accessing Nonlocal Environments

- Non local variables are those variables that are visible but not locally declared
  - Global variables are visible in all units
- Static environments (Fortran and COBOL)
  - All memory allocation can be performed at load time (static)
  - Location of variables fixed for the duration of program execution
  - Functions and procedures cannot be nested
  - Recursion is not allowed
- Stack-based environments
  - Block structured language with recursion $\rightarrow$ activation of procedure blocks cannot be allocated statically
  - A new activation record is created on the stack when a block is entered and is released on exit (return)
    - Space needs to be allocated for local variables, temporary space, and a return pointer
    - A dynamic link stores the old environment pointer
    - A static link points to the static parent (for non-local references)
  - Must keep a pointer to the current activation record (stack pointer, stored in a register)

Stack-Based Environment Example

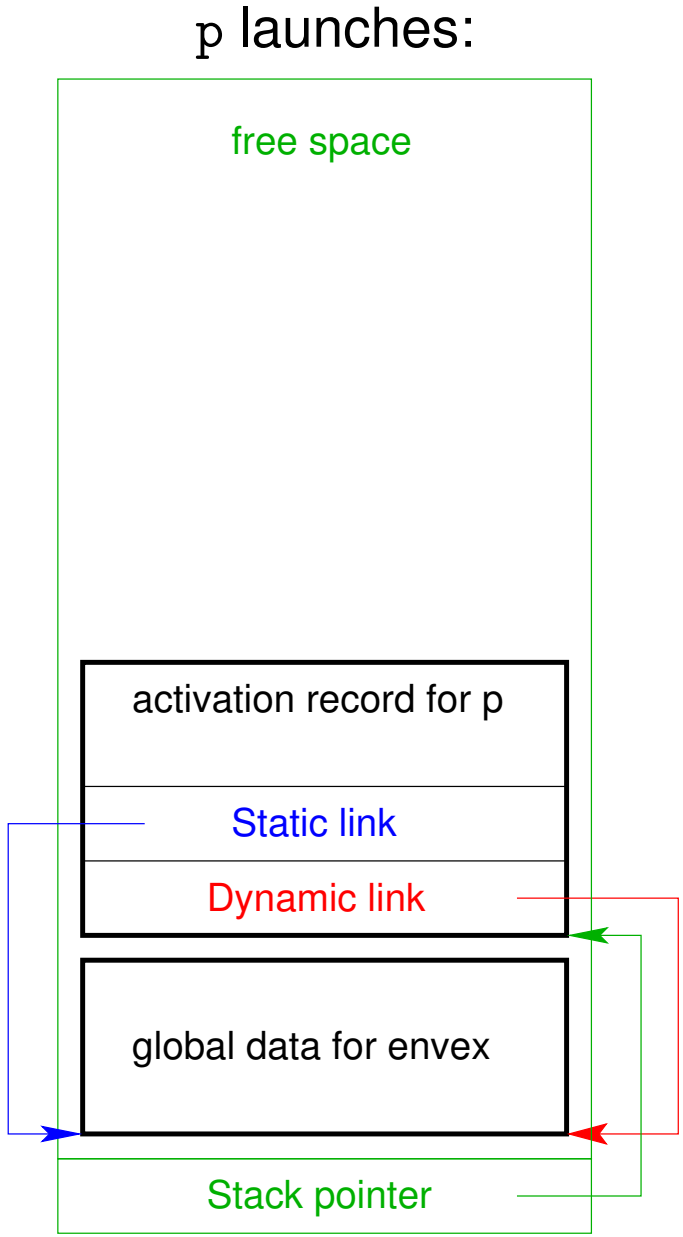
```
program envex:
   procedure q;
   begin
      ...
   end;
   procedure p;
   begin
      ...
      q;
      ...
   end
begin (*main*)
   ...
   p;
   ...
end.
```

Before `main` calls `p`:



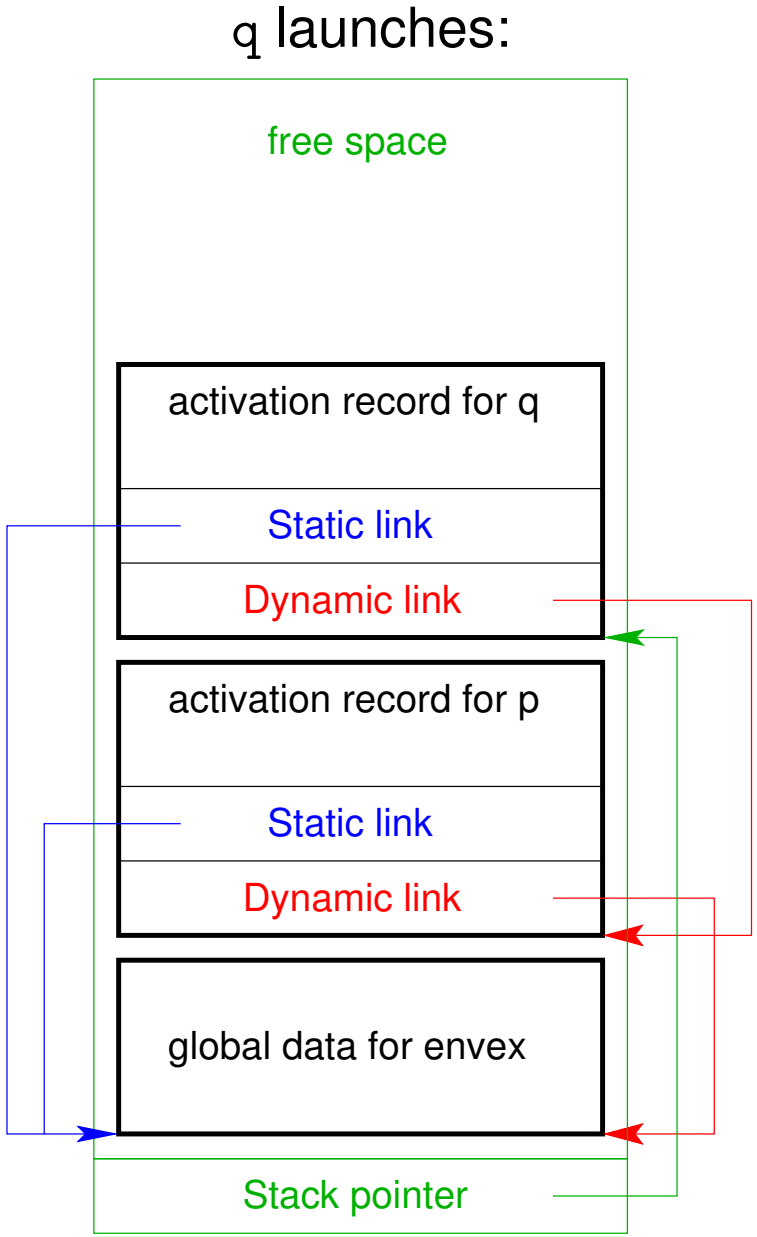
Stack-Based Environment Example (cont'd)

```
program envex:
    procedure q;
    begin
        ...
    end;
    procedure p;
    begin
        ...
        q;
        ...
    end
begin (*main*)
    ...
    p;
    ...
end.
```



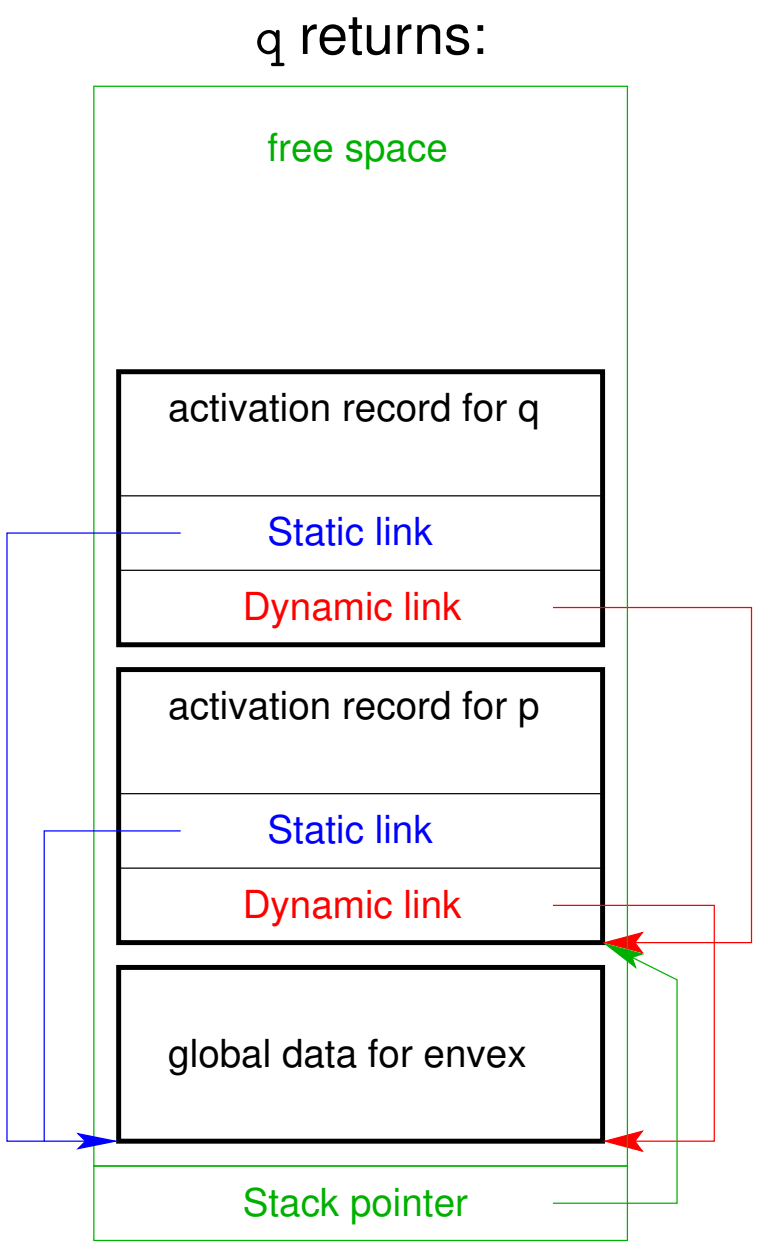
Stack-Based Environment Example (cont'd)

```
program envex:
    procedure q;
    begin
        ...
    end;
    procedure p;
    begin
        ...
        q;
        ...
    end
begin (*main*)
    ...
    p;
    ...
end.
```



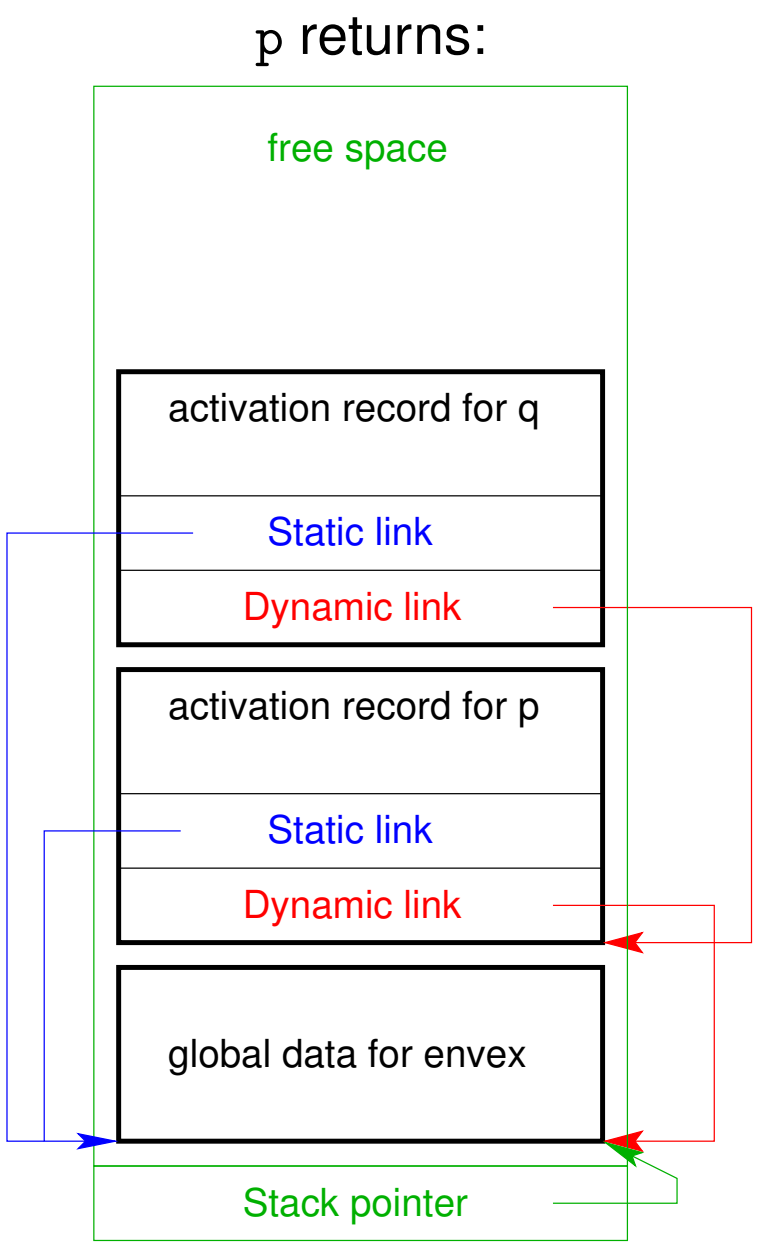
Stack-Based Environment Example (cont'd)

```
program envex:
    procedure q;
    begin
        ...
    end;
    procedure p;
    begin
        ...
        q;
        ...
    end
begin (*main*)
    ...
    p;
    ...
end.
```



Stack-Based Environment Example (cont'd)

```
program envex:
    procedure q;
    begin
        ...
    end;
    procedure p;
    begin
        ...
        q;
        ...
    end
begin (*main*)
    ...
    p;
    ...
end.
```



Content of the Activation Record

- Return address
  - Contains pointer back to code segment + offset of the address following the call
- Static link
  - Implements access to non-local variables for deep/lexical binding
  - Non-local references could be made by searching down the static chain
  - However, we cannot search at run time (no name information anymore!)
  - But scopes are known at compile time so the compuler knows the length of the static chain
  - Thus a non-local variable is represented by an ordered pair of integers `(chain_offset,local_offset)`
  - References to variables beyond static parent are costly
- Dynamic link
  - Represents the history of the execution
  - Implements access to non-local variables for shallow binding
- Parameters
- Local variables

Blocks

- Blocks can be treated as parameterless subprograms
  - Always called from same place
  - But we nonetheless need to access local as well as non-local variables
- The environment can be maintained in a stack-based fashion for any block structured language with static scoping