PROGRAMMING LANGUAGES RECITATION

-Monika Dagar

24th September 2020

Agenda

- Parameter Types
- Parameter Passing
- Stack Frame/Activation Record
- First Class Functions

Parameter Types

- Formal parameter (parameter): names / variables that appear in the declaration of the subroutine.
- Actual parameter (argument): the expressions passed to a subroutine at a particular call site.

Function:

```
int func1 (int a, char b, float& c)
 {
        ..
        ..
}
```

Formal Parameters

Function Call:

```
func1(5 * 3, 'A', z);
```

Actual Parameters

Parameter Passing

- Mechanism which determines what kind of association is in between the formal and actual parameters.
- Parameter passing techniques:
  - *By Value*
  - *By Reference*
  - *By Copy – Return (Value – Result)*
  - *By Name*
  - *By Need*

- Call by value
  - *Formal parameter (formal) is bound to copy of value of actual*
  - *If you do an assignment to formal, it only changes the value of formal, but not the actual.*
  - *Language specified by default: C/C++, Java.*
- Call by reference
  - *Formal is bound to location of actual, forming an alias*
  - *If you do an assignment to formal, it changes the value of actual.*
  - *Language supported: C++, C#.*
- *Call by copy – return*
  - *Formal is bound to value of actual*
  - *Upon return from routine, actual gets copy of formal.*
  - *in-out parameter mode in Ada*

- Call by name
  - *Formal is bound to the expression of actual*
  - *Expression is evaluated **each time** when the formal is used during execution*
  - *Algol 60 or more general principle*
    - If the actual parameter is a variable, this is the same as call by reference.
    - If the actual parameter is an expression, the expression is re-evaluated on each reference.
  - *Scala*
    - Formal parameter using call-by-name is immutable.
  - *Language supported: Algol 60, Scala.*
- Call by need
  - *Formal is bound to the expression of actual*
  - *Expression is evaluated **only once** when the formal is used at the first time.*
  - ***Subsequent reads** from the formal will use the value computed earlier.*
  - *If the actual parameter is a variable, this is the same as call by reference.*
  - *If the actual parameter is an expression, the expression is re-evaluated at the first access.*
  - *Language supported: Haskell, R.*

- Pass by value

global = 10

another = 2

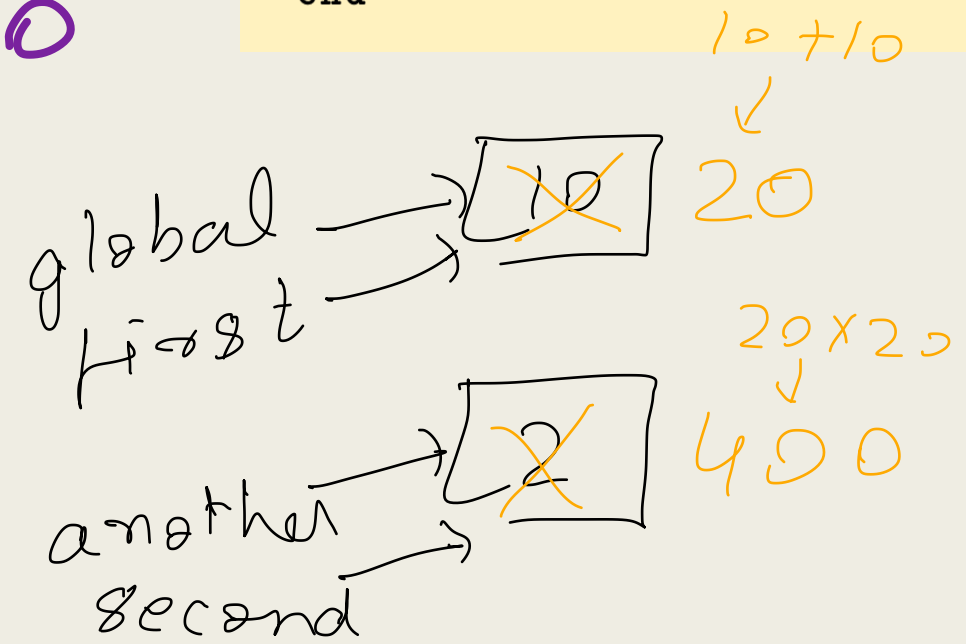
```
program example;
  var
    global: integer := 10;
    another: integer := 2;
  procedure confuse (var first, second: integer);
  begin
    first := first + global;
    second := first * global;
  end;
begin
  confuse(global, another);  /* first and global */
                             /*  are aliased     */
end
```

- Pass by reference

global = 20

another = 400

```
program example;
  var
    global: integer := 10;
    another: integer := 2;
  procedure confuse (var first, second: integer);
  begin
    first := first + global;
    second := first * global;
  end;
begin
  confuse(global, another);  /* first and global */
                             /*  are aliased     */
end
```



- Pass by copy return

global = 20

another = 200

```
program example;
  var
    global: integer := 10;
    another: integer := 2;
  procedure confuse (var first, second: integer);
  begin
    first := first + global;
    second := first * global;
  end;
begin
  confuse(global, another);  /* first and global */
                             /*  are aliased     */
end
```

- x is call-by-value and y is call-by-name

6

3

```
Int incr(Int& k) { // call by reference
        k = k + 1
        return k
}

Void f(Int x, Int y) {
  x = y + 1
  println(x + y)
}

Int z = 1
f(z, incr(z))
println(z)
```

- x is call-by-reference and y is call-by-name

7

4

```
Int incr(Int& k) { // call by reference
        k = k + 1
        return k
}

Void f(Int x, Int y) {
  x = y + 1
  println(x + y)
}

Int z = 1
f(z, incr(z))
println(z)
```

Performance considerations

- By Value
- By Reference
- By Copy – Return (Value – Result)
- By Name
- By Need

Stack Frame/Activation Record

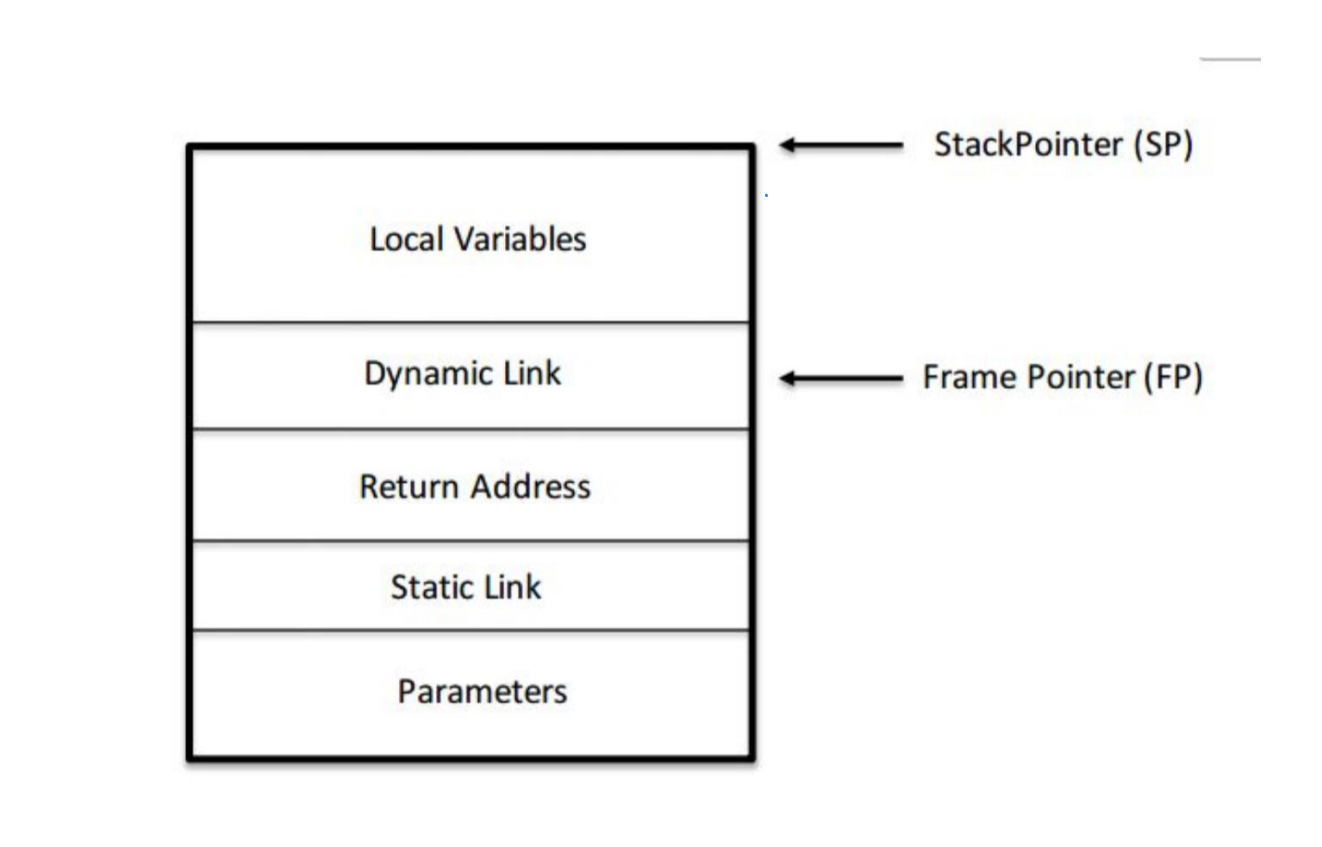
Stack

- A stack is a linear structure where data are inserted and deleted according to the last-in first-out (LIFO) principle.
- Operations:
  - *Push: Adds the element into the container.*
  - *Pop: Removes the most recently added element that was not yet removed.*

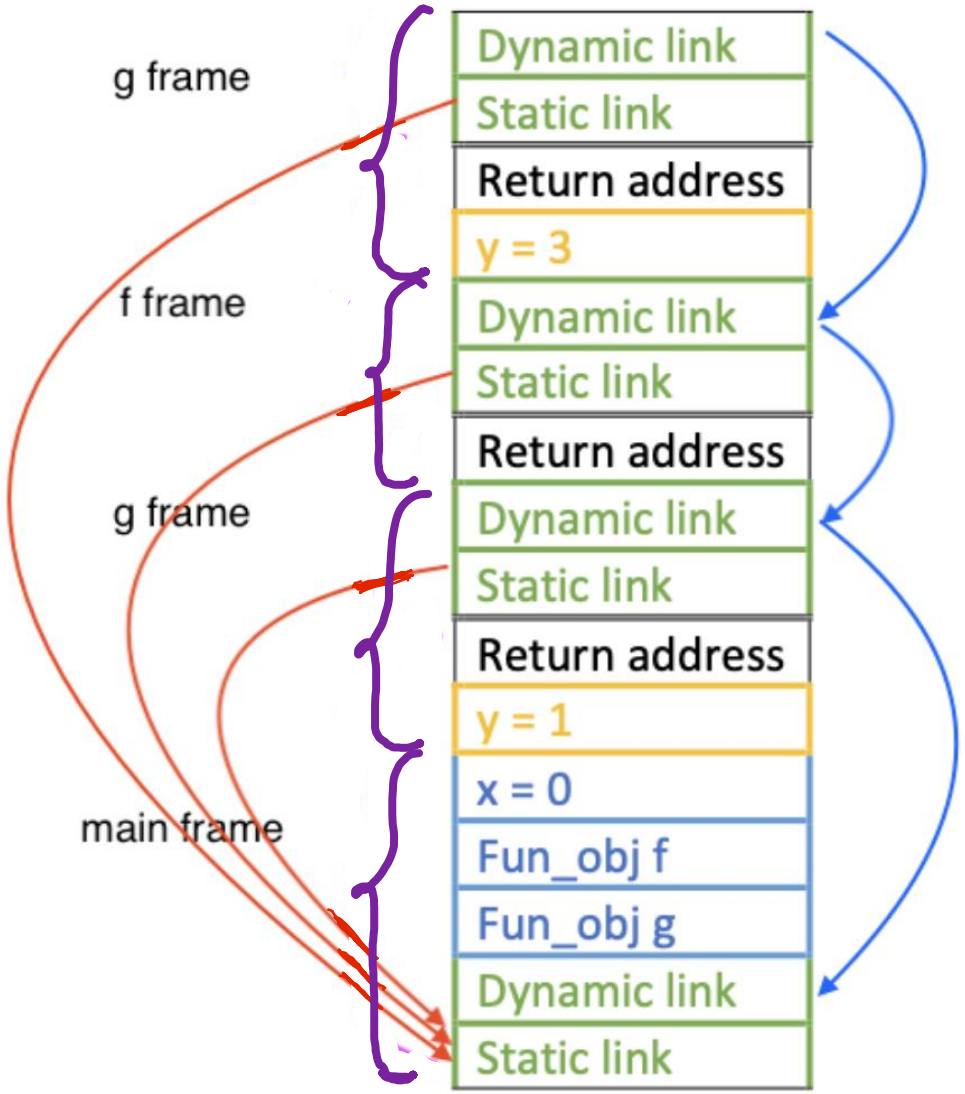
Activation Records / Stack Frames

- The piece of memory on the stack used for storing information of a particular function call.
- When a subroutine returns, its frame is popped from the stack

Layouts (in general):



- Stack Pointer (SP): points to the top of the stack. That is, it holds the address of the last item put on the stack.
- Return Address: store the return address that resumes at the point in the code after the call.
- Frame Pointer (FP): a pointer that points into the activation record of a subroutine so that any objects allocated on the stack can be referred with a static offset from the frame pointer.
- Local Data: a space to allow the subroutine to store local variables.
- Static Link: a pointer that points to the activation record of the lexically surrounding subroutine. The value that is stored here is the address of the frame of the procedure that statically encloses the current procedure.
- Dynamic Link: a pointer that points to the frame of the caller. The value that is stored here is the address of its caller's frame on the stack.




```
1: def main ():
2:     x = 10
3:     def f():
4:         print(x)
5:         g(3)
6:
7:     def g(y):
8:         if y == 1:
9:             f()
10:         else:
11:             print("exit!")
12:     g(1)
13:
14: main()
```

Ada

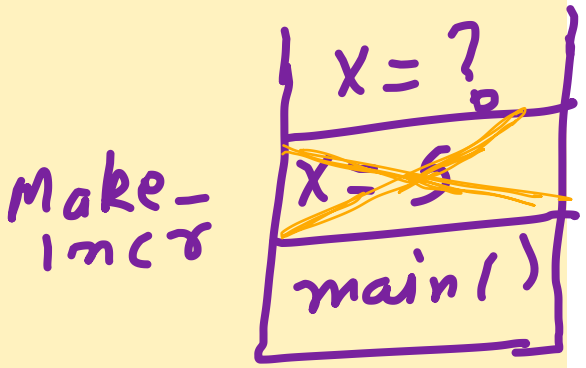
The limits of stack allocation

```
type Ptr is access function (X: Integer) return Integer;

function Make_Incr (X: Integer) return Ptr is
  function Incr (Base: Integer) return Integer is
  begin
    return Base + X;   -- reference to formal of Make_Incr
  end;
begin
  return Incr'access;   -- will it work?
end;

Add_Five: Ptr := Make_Incr(5);

Total: Integer := Add_Five(10);   -- where does Add_Five
                                  --  find X ?
```



First-class functions

- In some programming languages (usually functional languages), they treat functions as first-class citizen.
- Thus, those languages support a subroutine to pass a function as a parameter or to return a function.
- The question is if we pass a function or return a function, how could we remain the environment of that function declared.

```
def create_adder():
        i = 3 # Local variable
        return lambda x : i + x

adder = create_adder()
# print(i) error!
print(adder(2)) # 5
```

- Closure: a record storing function (reference) together with an environment.
- Environment: a mapping associating each free variable of the function with the value or reference to which the name was bound when the closure was created.
- Free variable: variable is not defined by the current function. ⇒ i