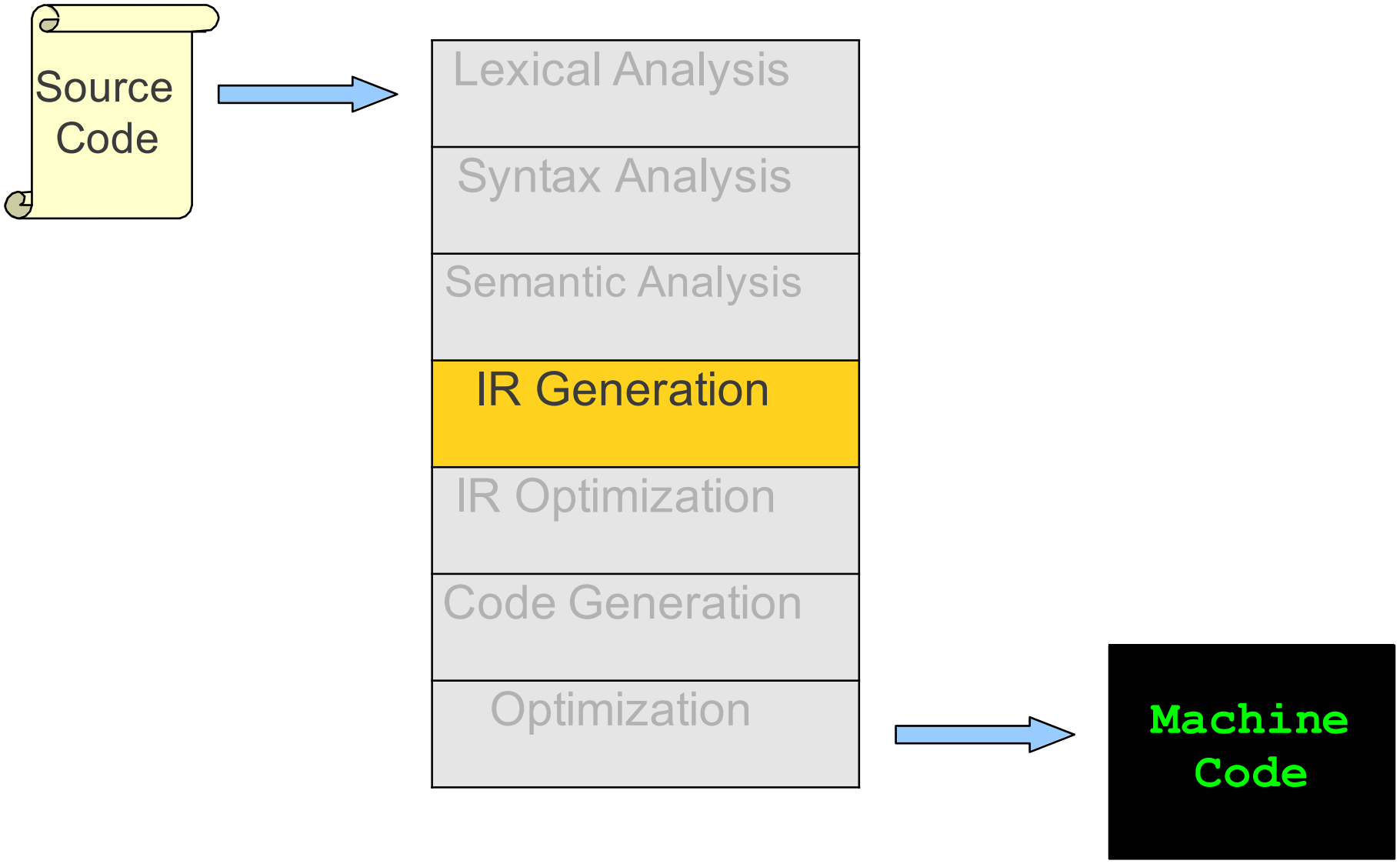
Run-time Environments

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

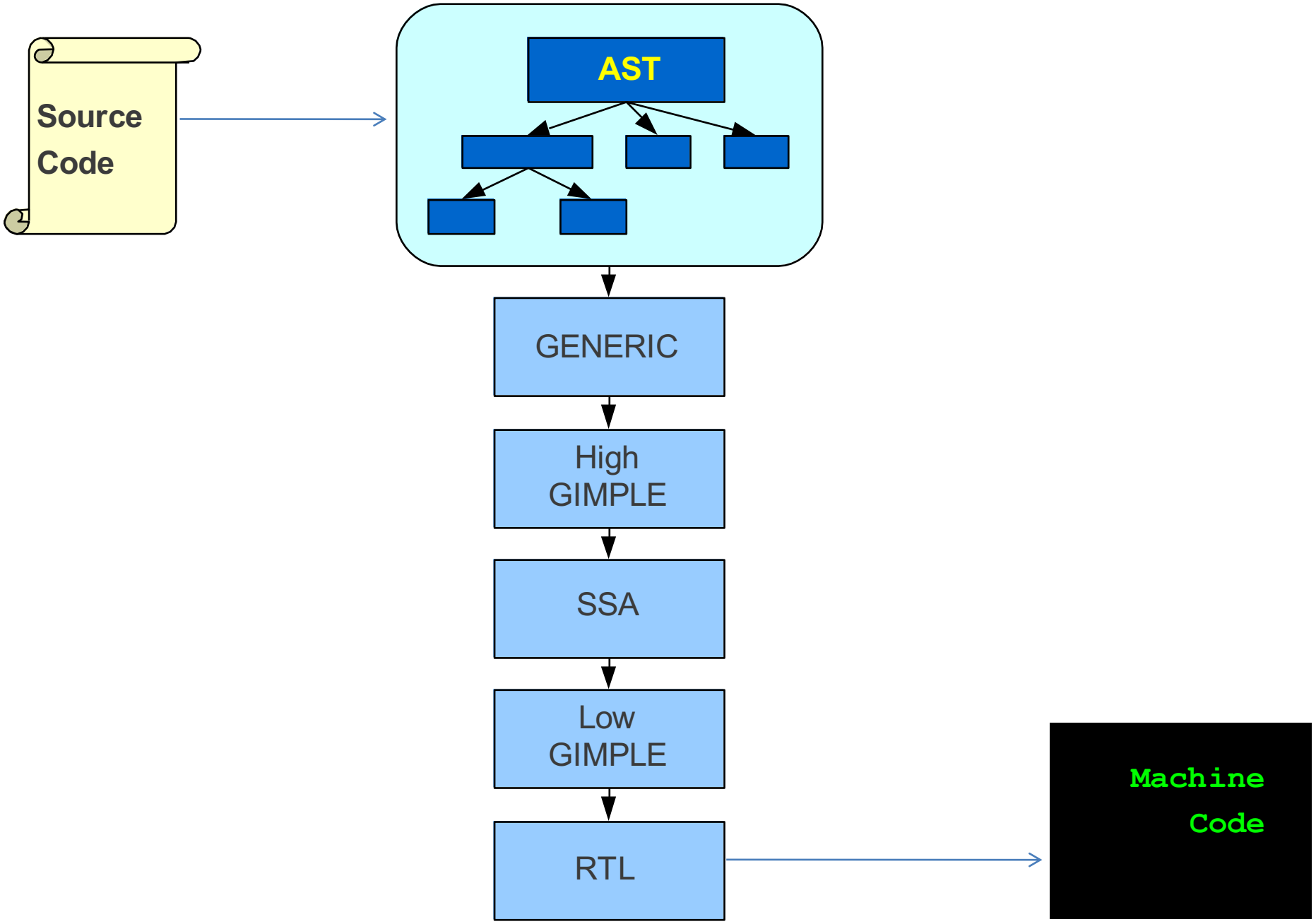
Why Do IR Generation?

- **Simplify certain optimizations.**
  - Machine code has many constraints that inhibit optimization. (Such as?)
  - Working with an intermediate language makes optimizations easier and clearer.
- **Have many front-ends into a single back-end.**
  - `gcc` can handle C, C++, Java, Fortran, Ada, and many other languages.
  - Each front-end translates source to the GENERIC language.
- **Have many back-ends from a single front-end.**
  - Do most optimization on intermediate representation before emitting code targeted at a single machine.

Designing a Good IR

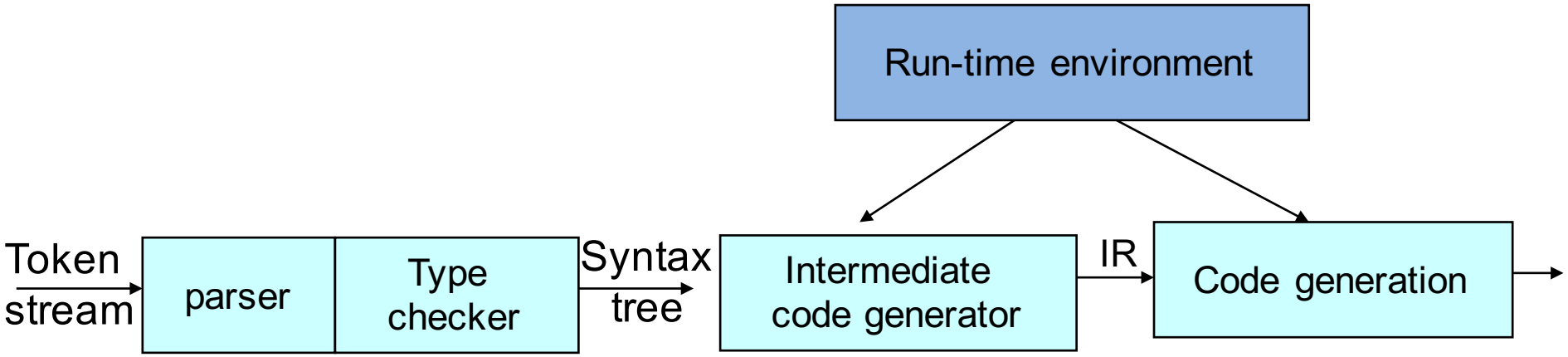
- IRs are like type systems – they're extremely hard to get right.
- Need to balance needs of high-level source language and low-level target language.
- Too high level: can't optimize certain implementation details.
- Too low level: can't use high-level knowledge to perform aggressive optimizations.
- Often have multiple IRs in a single compiler.

Architecture of gcc



Run-time Environments

- How do we implement language features in machine code?
- What data structures do we need?



Run-time Environments

- A compiler must accurately implement the abstractions embodied in the source language definition. These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters, and flow-of-control constructs.
- The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.
- To do so, the compiler creates and manages a **run-time environments** in which it assumes its target programs are being executed. This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters, and the interfaces to the operating system, input/output devices, and other programs.

Run-Time Environments (cont.)

- How do we allocate the space for the generated target code and the data object of our source programs?
- The places of the data objects that can be determined at compile time will be *allocated* ***statically***.
- But the places for the some of data objects will be *allocated at* ***run-time dynamically*****.**
- The allocation of de-allocation of the data objects is managed by the *run-time support package*.
  - run-time support package is loaded together with the generate target code.
  - the structure of the run-time support package depends on the semantics of the programming language (especially the semantics of procedures in that language).

An Important Duality

- Programming languages contain high-level structures:
  - Functions
  - Objects
  - Exceptions
  - Dynamic typing
  - Lazy evaluation
  - (etc.)
- The physical computer only operates in terms of several primitive operations
  - Arithmetic Data
  - Movement
  - Control jumps

```
   219: BYTE XtoEE(BYTE xdata * XAddr, WORD EEAddr, WORD Length)
Y:0x827534  900C81        MOV       DPTR,#XAddr(0x0C81)
Y:0x827537  EF            MOV       A,R7
Y:0x827538  A50E          MOVX      @DPTR,R6,A
Y:0x82753A  A3            INC       DPTR
Y:0x82753B  A3            INC       DPTR
Y:0x82753C  ED            MOV       A,R5
Y:0x82753D  A50C          MOVX      @DPTR,R4,A
Y:0x82753F  A3            INC       DPTR
Y:0x827540  A3            INC       DPTR
Y:0x827541  EB            MOV       A,R3
Y:0x827542  A50A          MOVX      @DPTR,R2,A
```

```
    55:          if((clock == 0xFF) || (clock == 0x00)) clock = 0xEA;
Y:0x828EA9  EF            MOV       A,R7
Y:0x828EAA  F4            CPL       A
Y:0x828EAB  6003          JZ        Y:0x828EB0
Y:0x828EAD  EF            MOV       A,R7
Y:0x828EAE  7002          JNZ       Y:0x828EB2
Y:0x828EB0  7FEA          MOV       R7,#CLMOD(0xEA)
```

Runtime Environments

- We need to come up with a representation of these high-level structures using the low-level structures of the machine.
- A **runtime environment** is a set of data structures maintained at runtime to implement these high-level structures.
  - e.g. the stack, the heap, static area, virtual function tables, etc.
- Strongly depends on the features of both the source and target language. (e.g compiler vs. cross- compiler)
- Our IR generator will depend on how we set up our runtime environment.

Data Representations

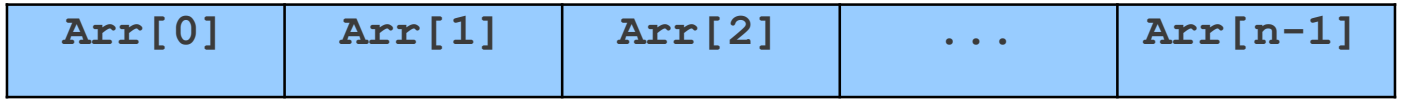
- What do different types look like in memory?
- Machine typically supports only limited types:
  - Fixed-width integers: 8-bit, 16-bit- 32-bit, signed, unsigned, etc.
  - Floating point values: 32-bit, 64-bit, 80-bit IEEE 754.
- How do we encode our object types using these types?

Encoding Primitive Types

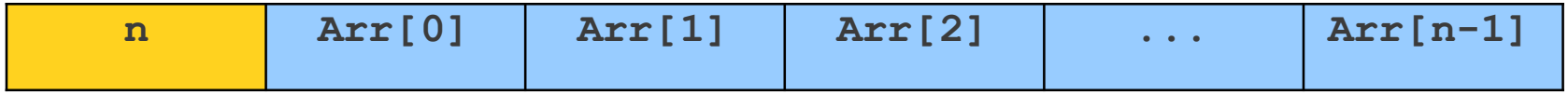
- Primitive integral types (**byte**, **char**, **short**, **int**, **long**, **unsigned**, **uint16_t**, etc.) typically map directly to the underlying machine type.
- Primitive real-valued types (**float**, **double**, **long double**) typically map directly to underlying machine type.
- Pointers typically implemented as integers holding memory addresses.
  - Size of integer depends on machine architecture; hence 32-bit compatibility mode on 64-bit machines.

Encoding Arrays

- C-style arrays: Elements laid out consecutively in memory.



- Java-style arrays: Elements laid out consecutively in memory with size information prepended.

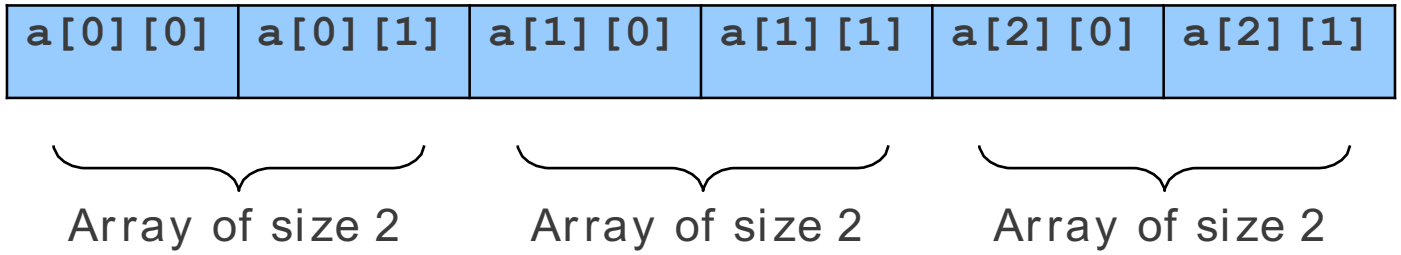


Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- C-style arrays:

```
int a[3][2];
```

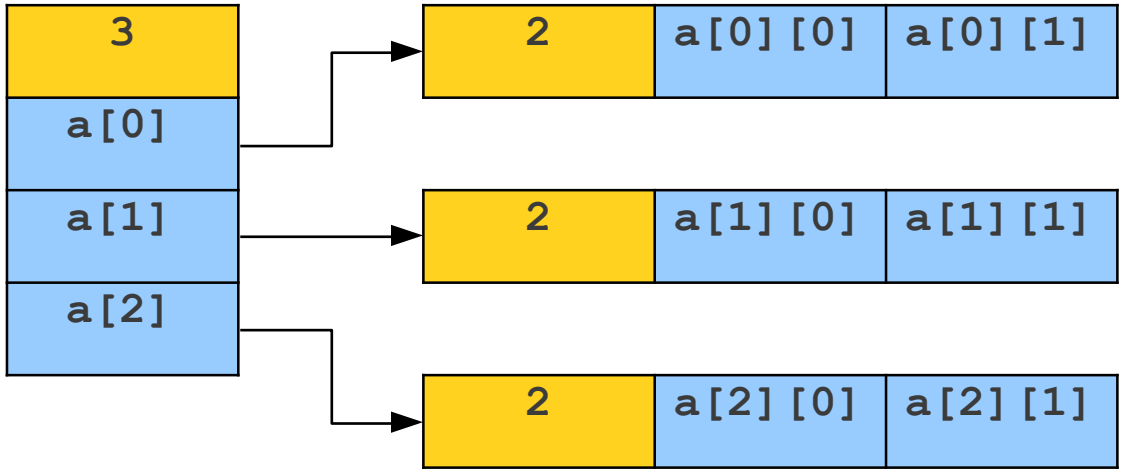
How do you know where to look for an element in an array like this?



Encoding Multidimensional Arrays

- Often represented as an array of arrays.
- Shape depends on the array type used.
- Java-style arrays:

```
int[][] a = new int [3][2];
```



Encoding Functions

- Many questions to answer:
  - What does the dynamic execution of functions look like?
  - Where is the executable code for functions located?
  - How are parameters passed in and out of functions?
  - Where are local variables stored?
- The answers strongly depend on what the language supports.

Procedure Activations

- An execution of a procedure starts at the beginning of the procedure body.
- When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
- Each execution of a procedure is called as its ***activation***.
- ***Lifetime*** of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
- If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested.
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.

Procedure Activations (cont.)

- Activation or liveness analysis is used for variables
- Functions or procedures have more complex activation behaviour
- Problem: functions can be recursive
  - –This means each function activation has to keep it's locals and parameters distinct

Activation Trees

- An activation of a function is a particular invocation of that function
- Each activation will have particular values for the function parameters
- Each activation can call another activation before it becomes inactive
- The sequence of function calls can be represented as an ***activation tree***

Activation Tree (cont.)

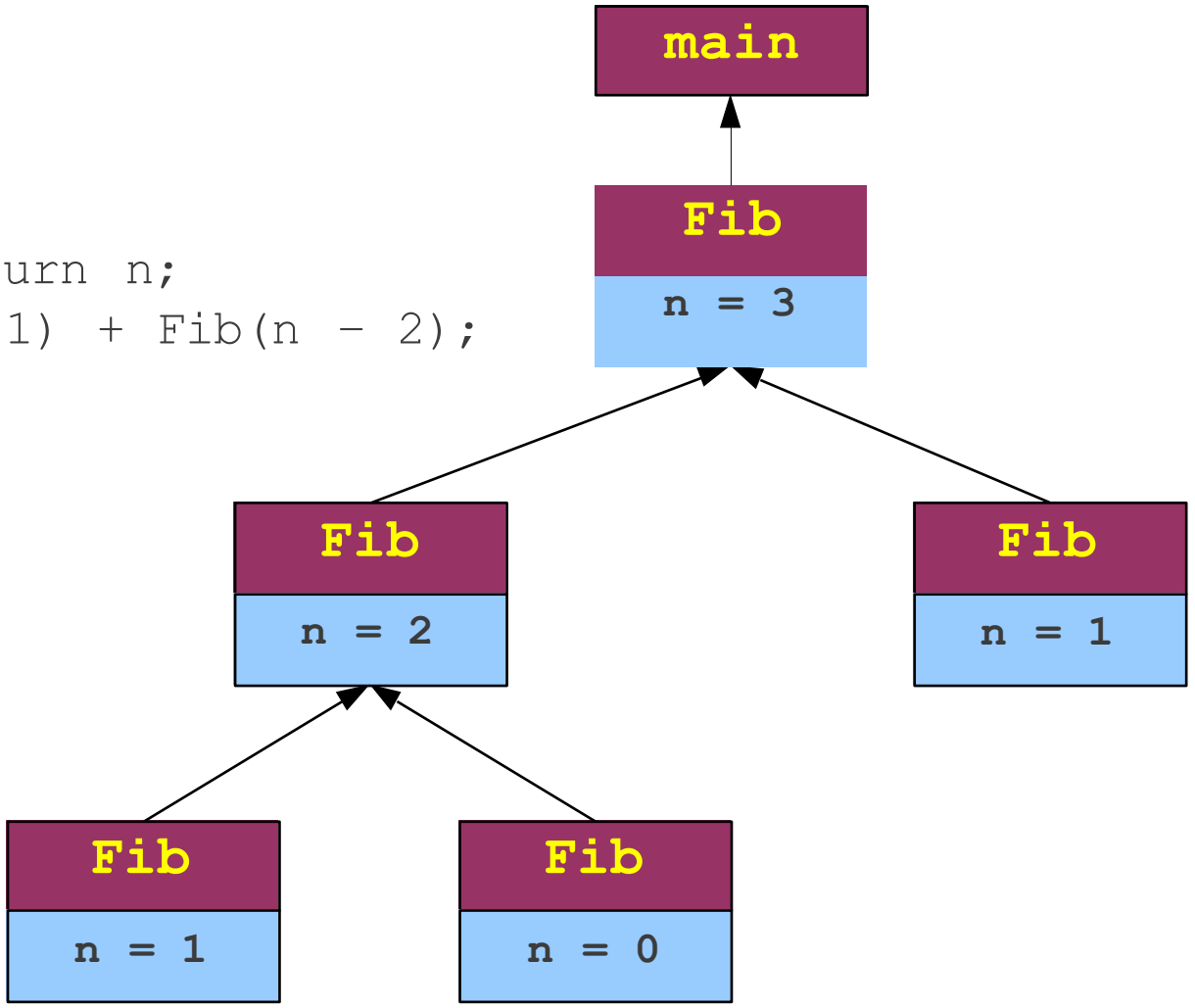
- We can use a tree (called **activation tree**) to show the way control enters and leaves activations.

- In an activation tree:
  - Each node represents an activation of a procedure.
  - The root represents the activation of the main program.
  - The node a is a parent of the node b iff the control flows from a to b.
  - The node a is left to the node b iff the lifetime of a occurs before the lifetime of b.

Activation Tree (eg)

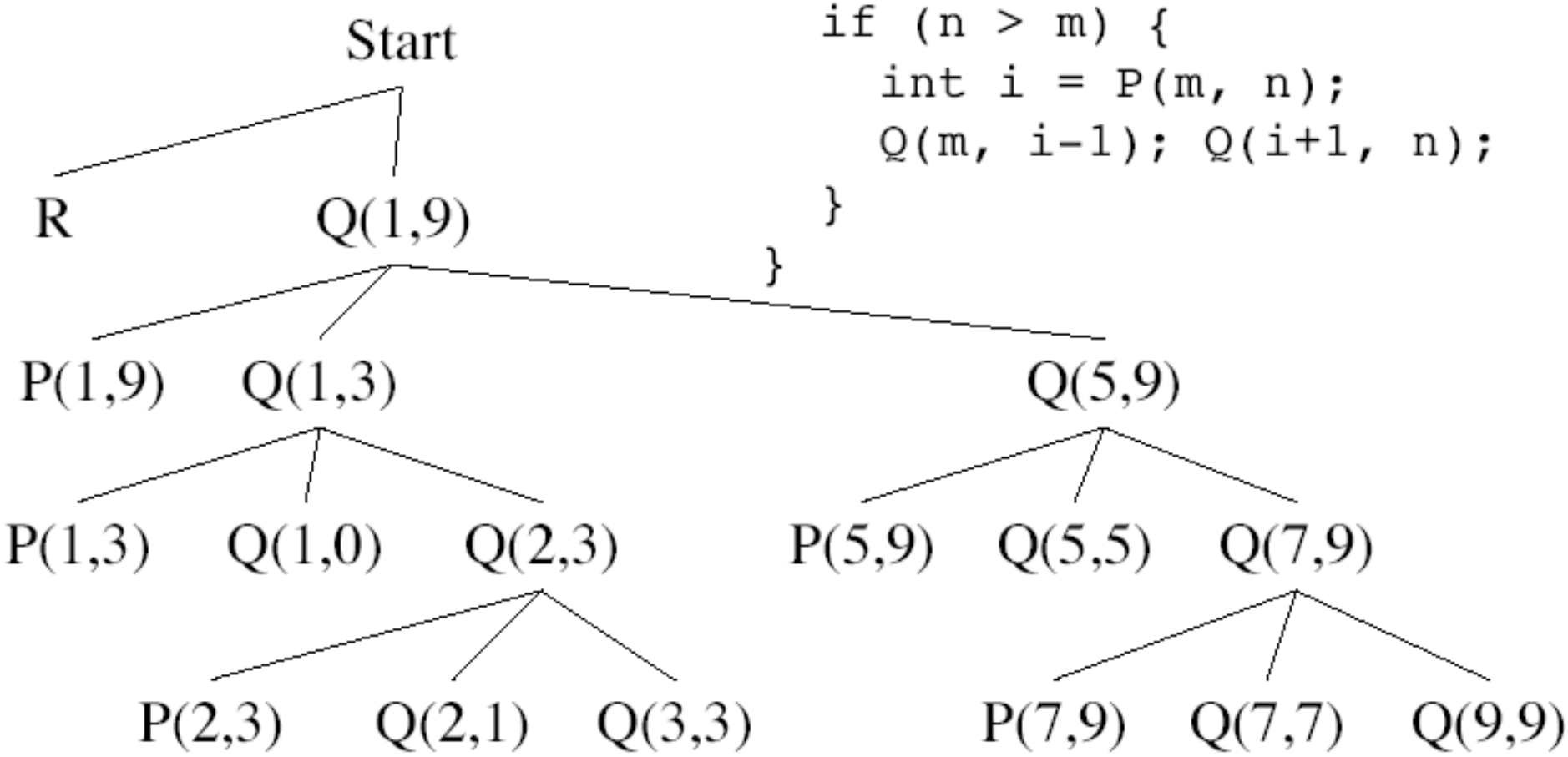
```
int main() {
    Fib(3);
}

int Fib(int n) {
    if (n <= 1) return n;
    return Fib(n - 1) + Fib(n - 2);
}
```



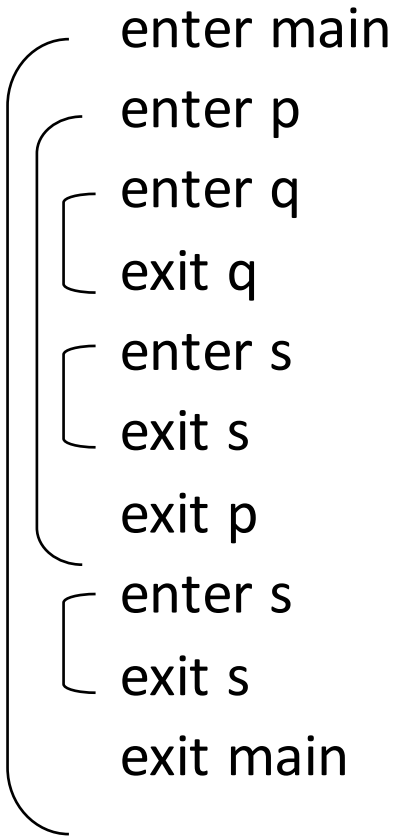
Activation Tree (cont.)

```
int Q(int m, int n) {
  if (n > m) {
    int i = P(m, n);
    Q(m, i-1); Q(i+1, n);
  }
}
```



Activation Tree (cont.)

```
program main;
  procedure s;
    begin ... end;
  procedure p;
    procedure q;
      begin ... end;
    begin q; s; end;
  begin p; s; end;
```



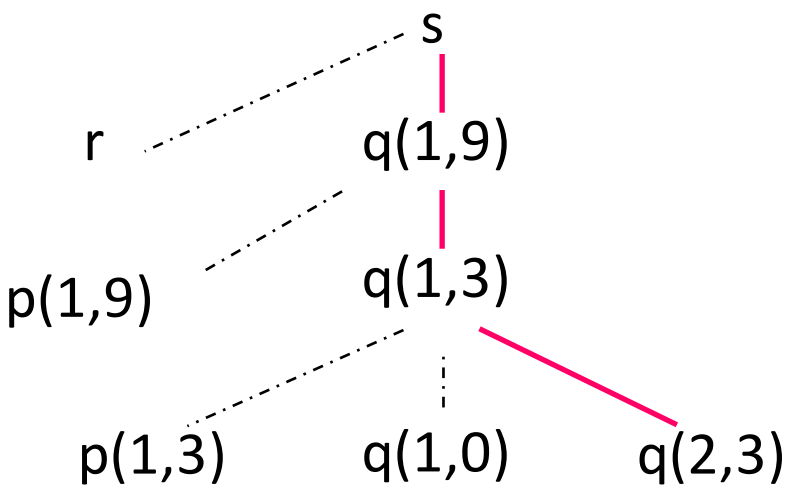
A Nested Structure

Control Stack

- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - starts at the root,
  - visits a node before its children, and
  - recursively visits children at each node an a left-to-right order.
- A stack (called **control stack**) can be used to keep track of live procedure activations.
  - An activation record is pushed onto the control stack as the activation starts.
  - That activation record is popped when that activation ends.
- When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

Control Stack (cont.)

- Control stack keeps track of live procedure activation.
- The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.



The control stack contains nodes along a path to the root.

An activation tree is a **spaghetti stack**.

The runtime stack is an **optimization** of this spaghetti stack.

Variable Scopes

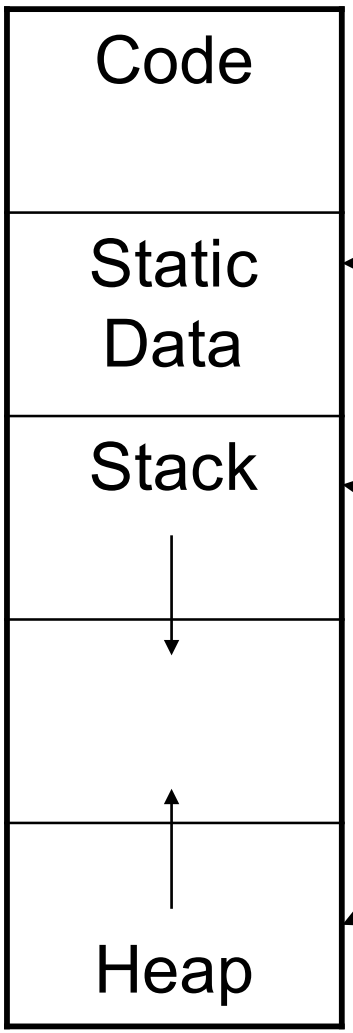
- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declaration of a name applies when the name appears in the program.
- An occurrence of a variable (a name) is:
  - **local**: If that occurrence is in the same procedure in which that name is declared.
  - **non-local**: Otherwise (ie. it is declared outside of that procedure)

```
procedure p;
  var b:real;
  procedure p;
    var a: integer;
    begin a := 1;  b := 2; end;
  begin ... end;
```

a is local

b is non-local

Run-Time Storage Organization



Memory locations for code are determined at compile time.

Locations of static data can also be determined at compile time.

Data objects allocated at run-time. (Activation Records)

Other dynamically allocated data objects at run-time. (For example, malloc area in C).

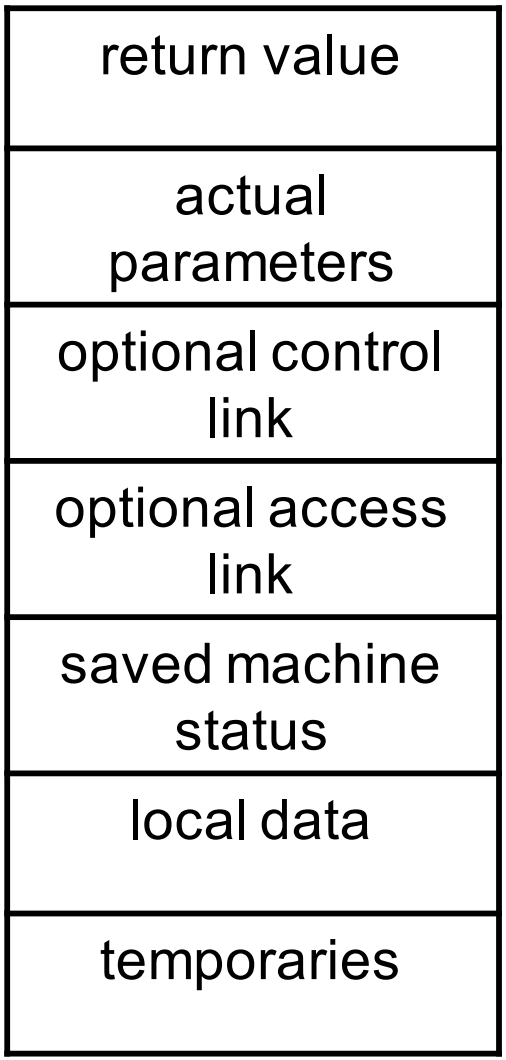
Problems with Functions

- If a function has local variables, and if it calls another function: what happens to locals after control returns
- Recursive functions
- Function can access non-local (global) variables
- Parameters passing into a function
- Can we pass functions as parameters?
- Can functions be returned as the result of a function?
- Storage allocation within a function?
- Is de-allocation to be done by the programmer before leaving the function?
- Dangling pointers?

Activation Records/Frame

- Information for a single execution of a function is called an ***activation record*** or ***procedure call frame***
- A frame contains:
  - –Temporary local register values for caller
  - –Local data
  - –Snapshot of machine state (important registers, PC)
  - –Return address (control link)
  - –Link to global data (access link)
  - –Parameters passed to function (actual parameters)
  - –Return value for the caller

Activation Records (cont.)



The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

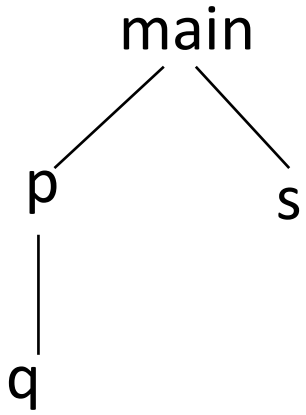
The field for saved machine status holds information about the state of the machine before the procedure is called.

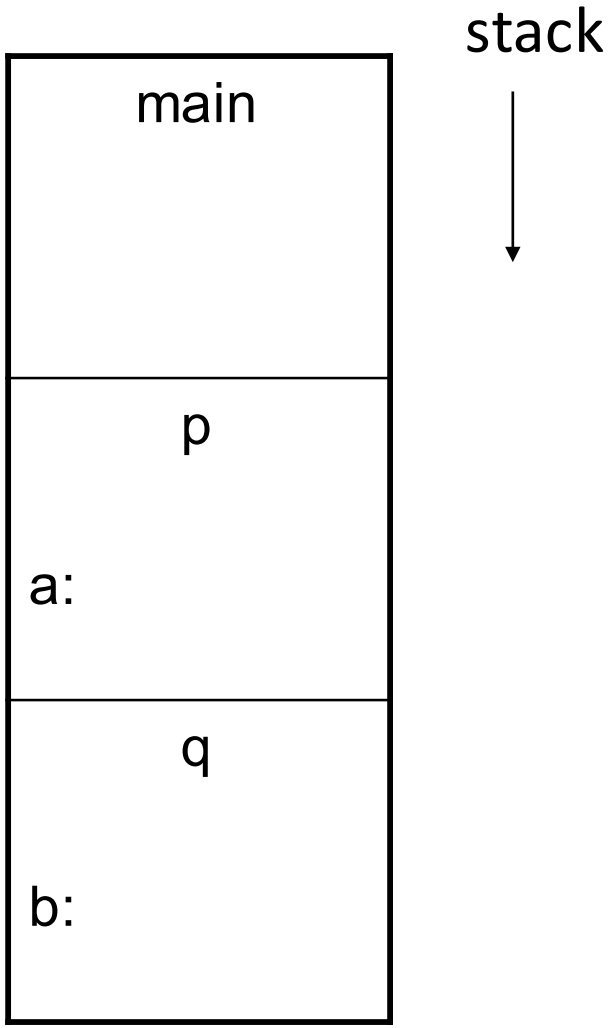
The field of local data holds data that local to an execution of a procedure..

Temporay variables is stored in the field of temporaries.

Activation Records (Ex1)

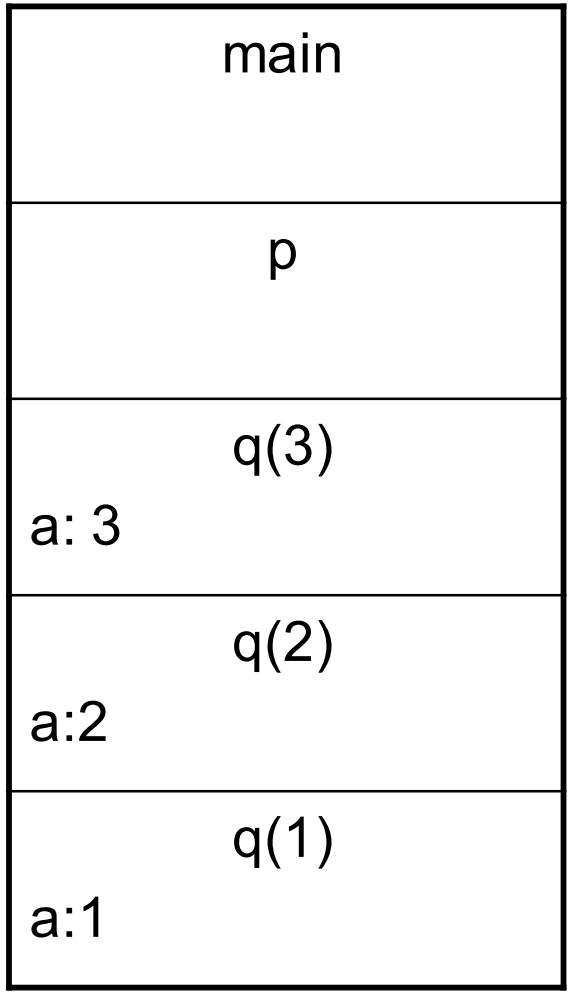
```
program main;
  procedure p;
    var a:real;
    procedure q;
      var
b:integer;
      begin ...
end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```





Activation Records for Recursive Procedures

```
program main;
  procedure p;
    function q(a:integer):integer;
      begin
        if (a=1) then q:=1;
        else q:=a+q(a-1);
      end;
    begin q(3); end;
  begin p; end;
```

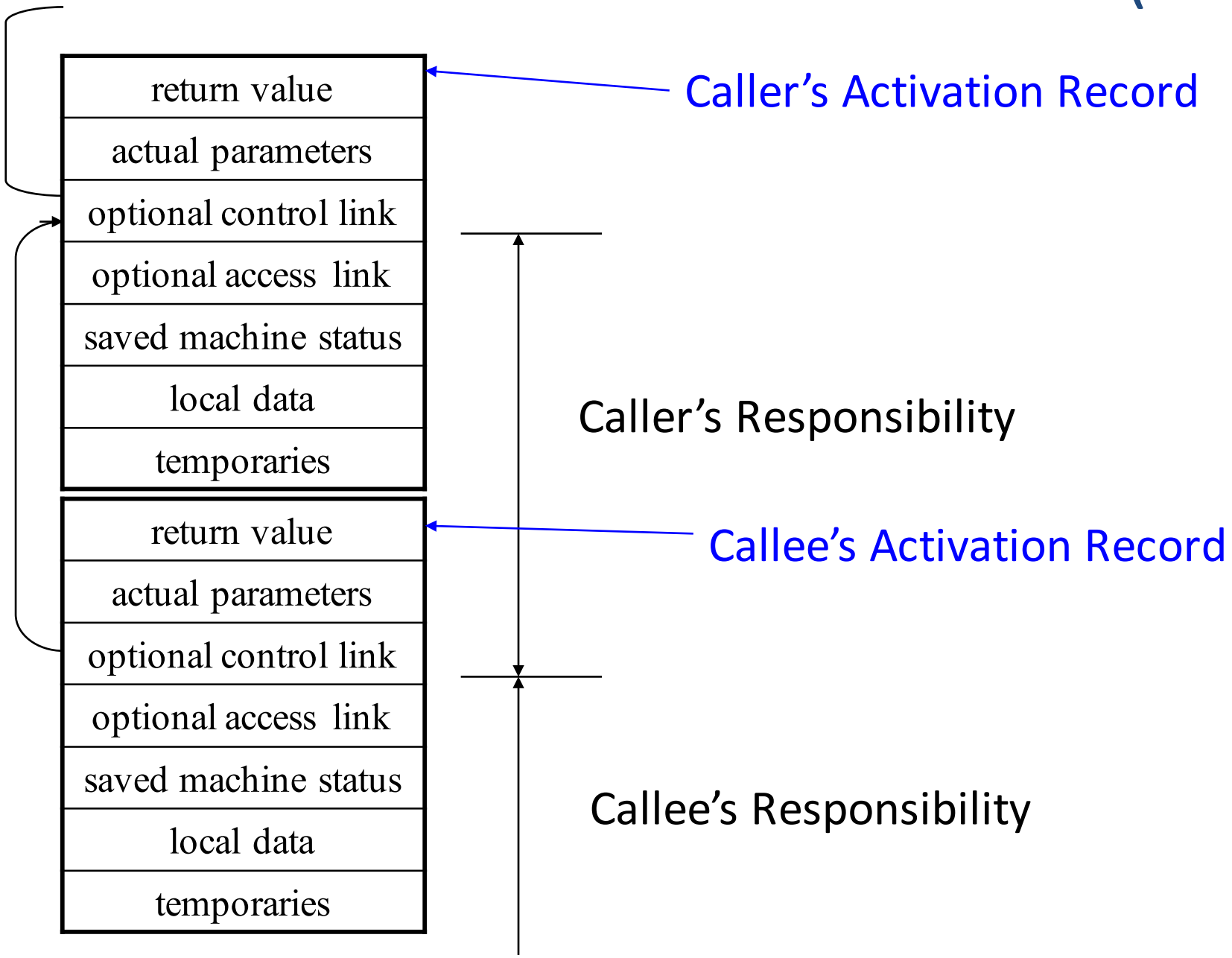


Creation of An Activation Record

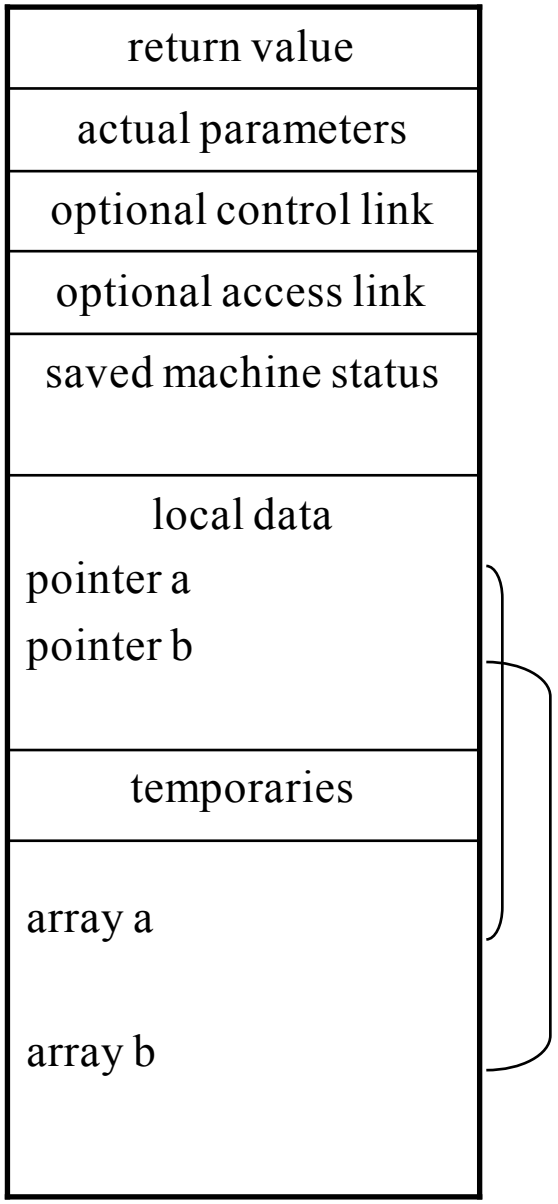
- Who allocates an activation record of a procedure?
  - Some part of the activation record of a procedure is created by that procedure immediately after that procedure is entered.
  - Some part is created by the caller of that procedure before that procedure is entered.

- Who deallocates?
  - Callee de-allocates the part allocated by Callee.
  - Caller de-allocates the part allocated by Caller.

Creation of An Activation Record (cont.)



Variable Length Data



Variable length data is allocated after temporaries, and there is a link to from local data to that array.

Access to Nonlocal Names

- Scope rules of a language determine the treatment of references to nonlocal names.
- Scope Rules:
  - **Lexical Scope (Static Scope)**
    - Determines the declaration that applies to a name by examining the program text alone at compile-time.
    - Most-closely nested rule is used.
    - Pascal, C, ..
  - **Dynamic Scope**
    - Determines the declaration that applies to a name at run-time.
    - Lisp, APL, ...

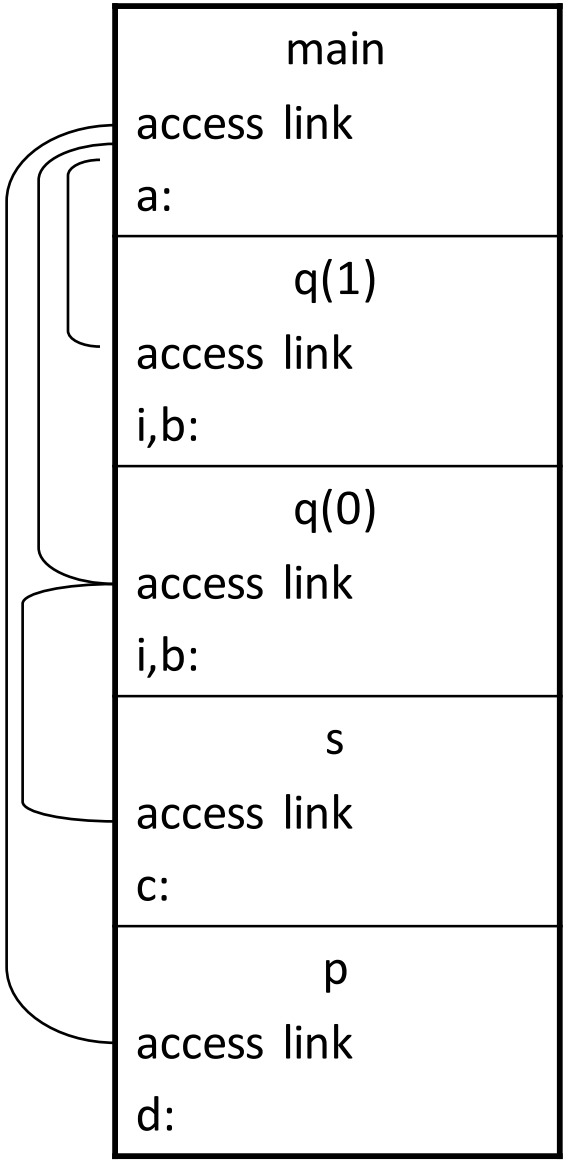
Lexical Scope

- The scope of a declaration in a block-structured language is given by the *mostly closed rule*.
- Each procedure (block) will have its own activation record.
  - –procedure
  - –begin-end blocks
    - (treated same as procedure without creating most part of its activation record)
- A procedure may access to a nonlocal name using:
  - –access links in activation records, or
  - –displays (an efficient way to access to nonlocal names)

Access Links

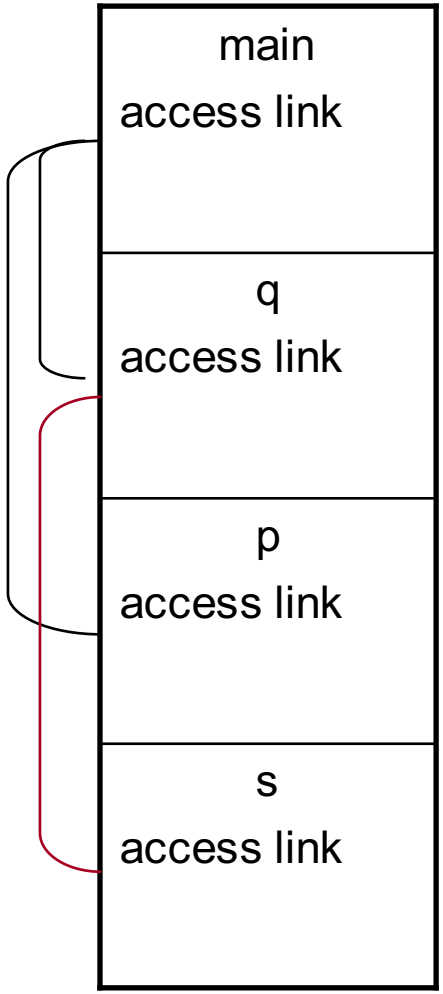
```
program main;
  var a:int;
  procedure p;
    var d:int;
    begin a:=1; end;
  procedure q(i:int);
    var b:int;
    procedure s;
      var c:int;
      begin p; end;
    begin
      if (i<>0) then q(i-1)
      else s;
    end;
  begin q(1); end;
```

Access Links



Procedure Parameters

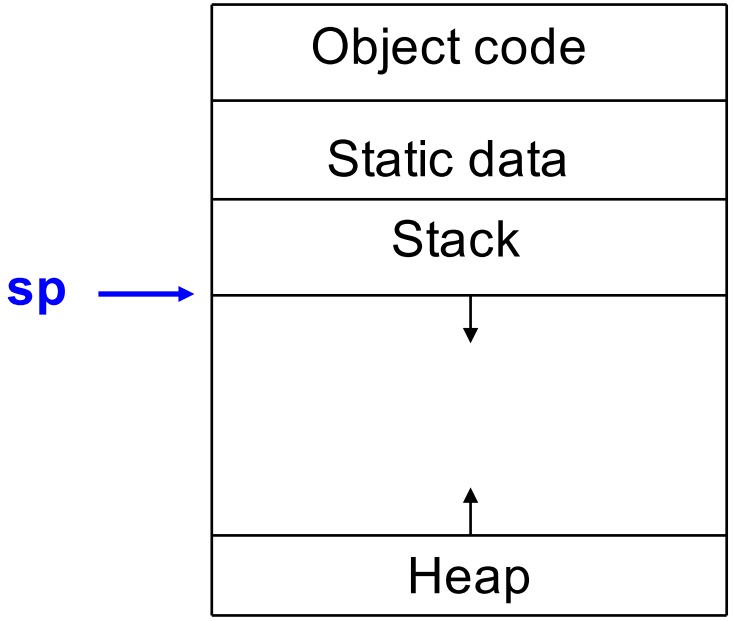
```
program main;
  procedure p(procedure a);
    begin a; end;
  procedure q;
    procedure s;
      begin ... end;
    begin p(s) end;
  begin q; end;
```



Access links must be passed with procedure parameters.

Storage organization

- The run-time storage might be subdivided:
  - –Target code
  - –Data object
  - –Control stack
  - –Heap



Typical subdivision of run-time memory into code and data areas.

Storage-allocation strategies

- Three allocation strategies:
  - **Static allocation** lays out storage for all data objects at compile time.
  - **Stack allocation** manages the run-time storage as a stack.
  - **Heap allocation** allocates and de-allocates storage as needed at run time from a data area known as a heap.

Storage Allocation for Functions

- Static Allocation
  - Layout all storage for all data objects at compile time
  - Essentially every variable is stored globally
  - But the symbol table can still control local activation and de-activation of variables
  - Very restricted recursion is allowed
  - Example: Fortran 77 program

Storage Allocation for Functions

- Stack Allocation
  - Activation records are associated with each function activation
  - Activation records are pushed onto the stack when a call is made to the function
  - Storage for recursive functions is organized as a stack: **last-in first-out (LIFO)** order

Storage Allocation for Functions

- Stack Allocation
  - Sometimes a minimum size is required
  - Variable length data is handled using pointers
  - Storage for the locals is contained in the activation record for that call.
  - Locals are deleted after activation ends
  - Caller locals are reinstated and execution continues
  - Example: C, Pascal and most modern programming languages

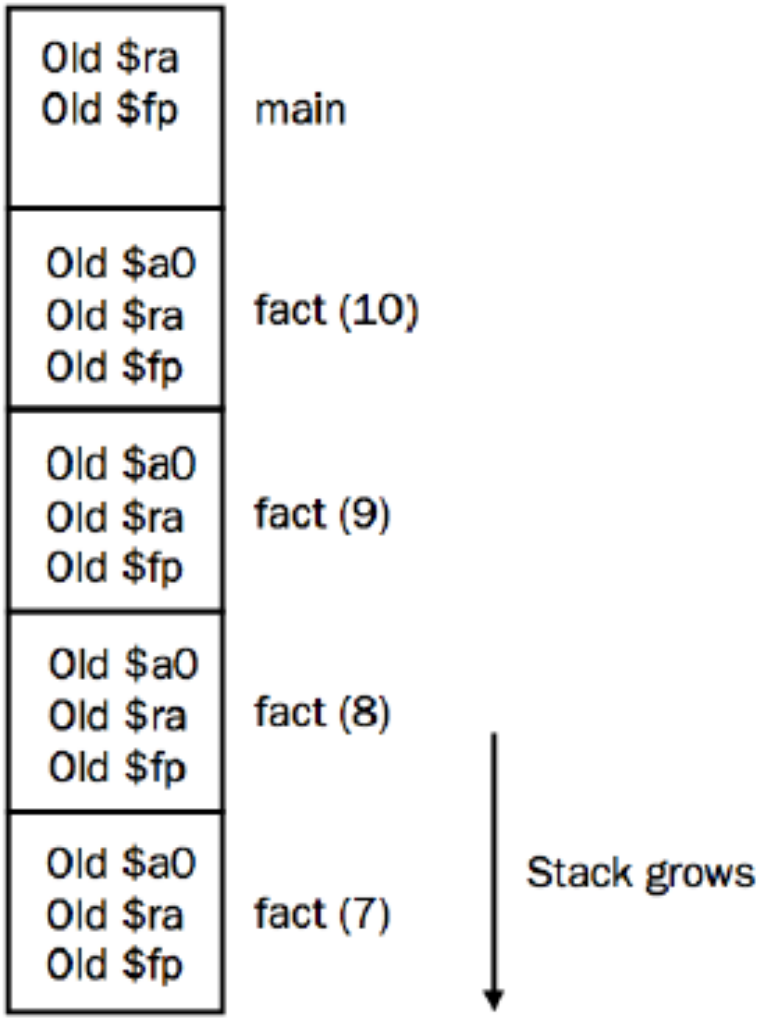
Storage Allocation for Functions

- Heap Allocation
  - In some special cases stack allocation is not possible
  - If local variables must be retained after the activation ends
  - If called activation outlives the caller
  - Anything that violates the last-in first-out nature of stack allocation e.g. functions as return values

Stack for recursive functions

Recursive function is used to calculate factorial of n!:

fact(n)=n*fact(n-1)



Parameter Passing Conventions

- Differences based on:
  - –The parameter represents an r-value (the rhs of an expr)
  - –An l-value
  - –Or the text of the parameter itself
- Call by Value
  - –Each parameter is evaluated
  - –Pass the r-value to the function
  - –No side-effect on the parameter
  - –The caller places the actual parameters (r-values) in the storage for the formals

Parameter Passing Conventions

- Call by Reference
  - –Also called call by address/location
  - –If the parameter is a name or expr that is an lvalue
  - –then pass the l-value
  - –else create a new temporary l-value and pass that
  - –Typical example: passing array elements a[i]
  - –The caller passes the called procedure a pointer to the storage address of each actual parameter

```
Main(){}
 A=2;  B=3;
 P(A+B, A, A);
 print A;
}
```

```
P(X, Y, Z){
    Y = Y+1;
    Z = Z+X;
}
```

Parameter Passing	Call by Value	Call by Reference
procedure	T: A+B=5	X=T=5, Y=Z=A=2 A=A+1=2+1 A=A+T=3+5
Result	2	8

Parameter Passing Conventions

- Copy Restore Linkage
  - Pass only r-values to the called function (but keep the lvalue around for those parameters that have it)
  - When control returns back, take the r-values and copy it into the l-values for the parameters that have it
  - Some Fortran implementations
- Call by Name
  - Function is treated like a macro (a #define) or in-line expansion
  - The parameters are literally re-written as passed arguments (keep caller variables distinct by renaming)

Parameter Passing Conventions

- Lazy evaluation
  - In some languages, call-by-name is accomplished by sending a function (also called a thunk) instead of an rvalue
  - When the r-value is needed the function is called with zero arguments to produce the r-value
  - This avoids the time-consuming evaluation of r-values which may or may not be used by the called function (especially when you consider short-circuit evaluation)
  - Used in lazy functional languages

Parameter Passing Conventions

- Call-by-need
  - Similar to lazy evaluation, but more efficient
  - To avoid executing similar r-values multiple times, some languages used a memo slot to avoid repeated function evaluations
  - A function parameter is only evaluated when used inside the called function
  - When used multiple times there is no overhead due to the memo table
  - Haskell

Summary

- Run-time Organization
  - The runtime environment has static data areas for the object code and the static data objects created at compile time.
- Memory Management
  - Static
  - Stack
  - Heap
- Control Stack
  - Procedure calls and returns are usually managed by a runtime stack called the control stack.
  - activation record
- Parameters Passing
  - Call by value
  - Call by reference
  - Call by name
  - Call by needed
  - (etc.)

Summary

- The runtime stack is an optimization of the activation tree spaghetti stack.
- Most languages use a runtime stack, though certain language features prohibit this optimization.
- Activation records logically store a **control link** to the calling function and an **access link** to the function in which it was created.
- Call-by-value and call-by-name can be implemented using copying and pointers.
- More advanced parameter passing schemes exist!

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu