# Activation Records

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc12-13.html

Chapter 6.3

# Outline of this lecture

- Operations on routines
- Stack Frames
- The Frame Pointer and Frame Size
- The Lexical Pointers and Nesting Levels
- Machine Architectures
- Parameter Passing and Return Address
- Frame Resident Variables
- Limitations
- Summary

# Operations on Routines

- Declarations
- Definitions
- Call
- Return
- Jumping out of routines
- Passing routines as parameters
- Returning routines as parameters

# Nested routines in C syntax

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int l;

            ...         /* code has access to i, j, k, l */
            k = l;
            j = l;
        }

        ...             /* code has access to i, j, k */
        j = k;
    }

    ...                 /* code has access to i, j */
}
```

# Non-Local goto in C syntax

```
void level_0(void) {

    void level_1(void) {

        void level_2(void) {

            ...
            goto L_1;
            ...
        }

        ...
    L_1:...
        ...
    }

    ...
}
```

# Non-local gotos in C

- setjmp remembers the current location and the stack frame
- longjmp jumps to the current location (popping many activation records)

# Non-Local Transfer of Control in C

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

# Passing a function as parameter

```
void foo (void (*interrupt_handler)(void))
{
 …
  if (…)  interrupt_handler();
  …
}
```

# Currying in C syntax

```
int (*)() f(int x)
{
   int g(int y)
   {
       return x + y;
    }
    return g ;
 }

int (*h)() = f(3);
int (*j)()  = f(4);

int z = h(5);
int  w = j(7);
```

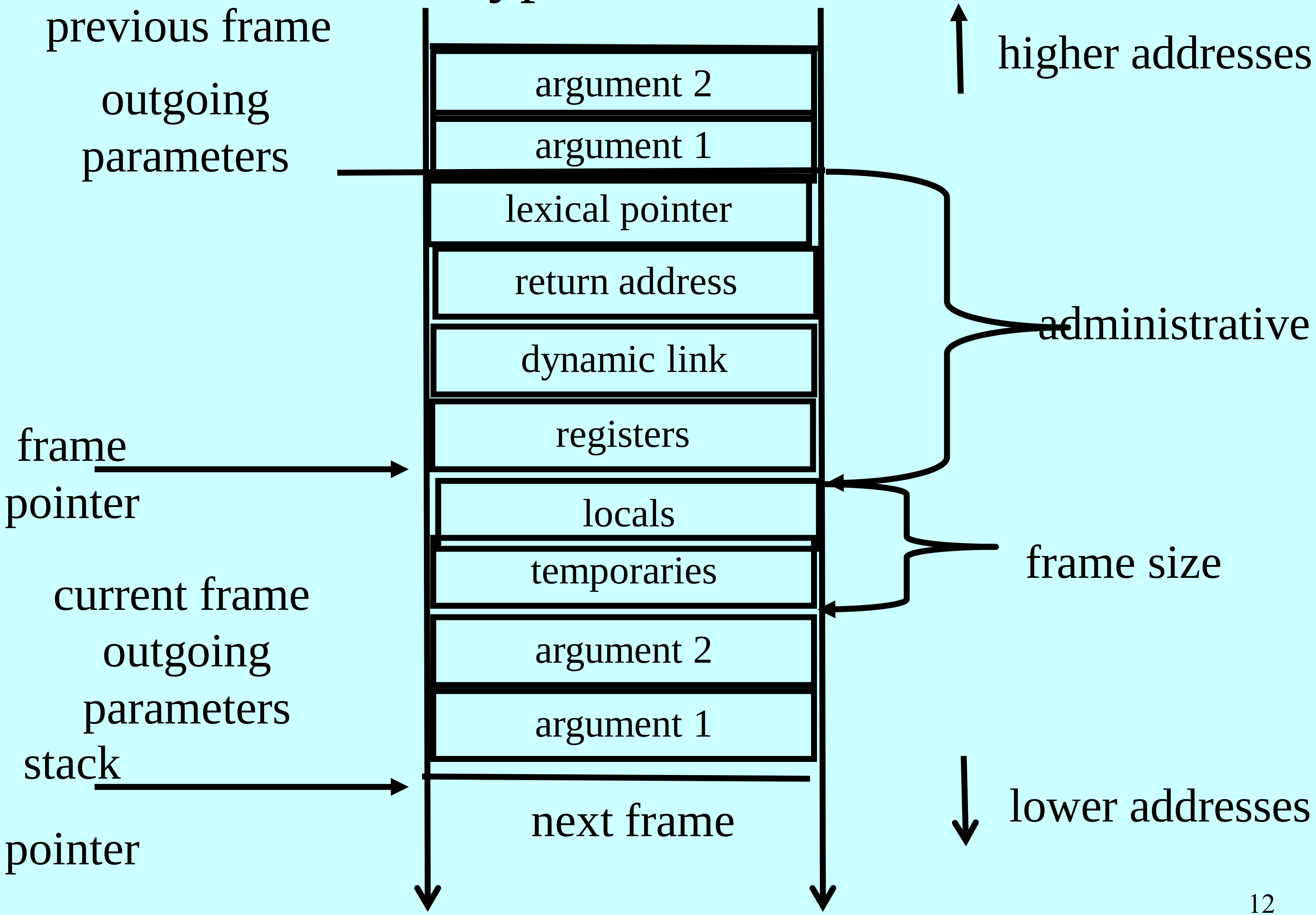
# Compile-Time Information on Variables

- Name
- Type
- Scope
  - when is it recognized
- Duration
  - Until when does its value exist
- Size
  - How many bytes are required at runtime
- Address
  - Fixed
  - Relative
  - Dynamic

# Stack Frames

- Allocate a separate space for every procedure incarnation
- Relative addresses
- Provide a simple mean to achieve modularity
- Supports separate code generation of procedures
- Naturally supports recursion
- Efficient memory allocation policy
  - Low overhead
  - Hardware support may be available
- LIFO policy
- Not a pure stack
  - Non local references
  - Updated using arithmetic

# A Typical Stack Frame



# L-Values of Local Variables

- The offset in the stack is known at compile time
- L-val(x) = FP+offset(x)
- x = 5 $\Rightarrow$ Load_Constant 5, R3
  Store R3, offset(x)(FP)

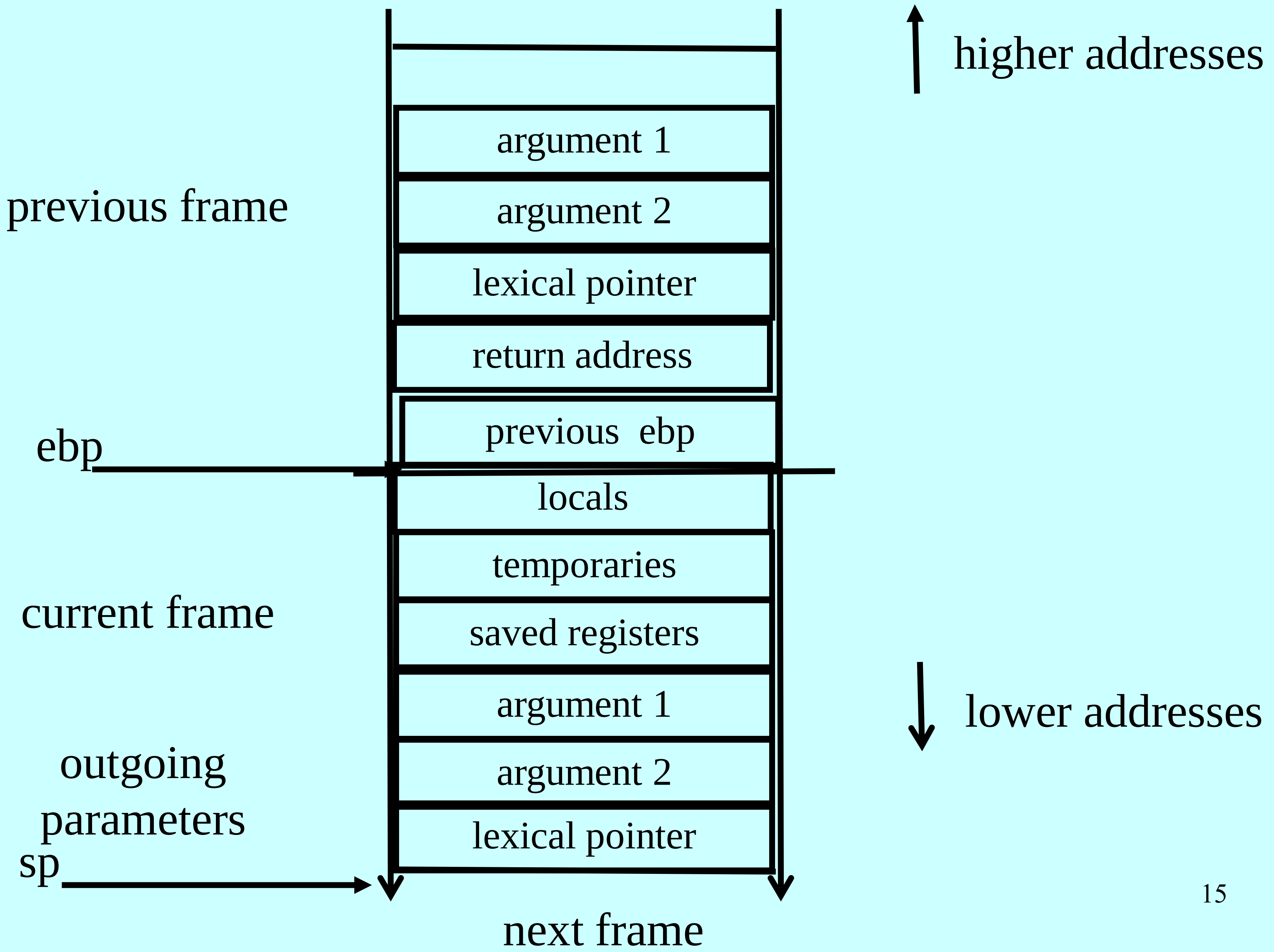
# Code Blocks

- Programming  language provide code blocks

```
void foo()
{
  int x = 8 ; y=9;//1
    { int x = y * y ;//2 }
    { int x = y * 7 ;//3}
        x = y + 1;
 }
```

| **adminstrative** |
| --- |
| x1 |
| y1 |
| x2 |
| x3 |
| … |

# Pascal 80386 Frame



# Summary thus far

- The structure of the stack frame may depend on
  - Machine
  - Architecture
  - Programming language
  - Compiler Conventions
- The stack is updated by:
  - Emitted compiler instructions
  - Designated hardware instructions

# The Frame Pointer

- The caller
  - the calling routine
- The callee
  - the called routine
- caller responsibilities:
  - Calculate arguments and save in the stack
  - Store lexical pointer
- call instruction:
  M[--SP] := RA
  PC := callee
- callee responsibilities:
  - FP := SP
  - SP := SP - frame-size
- Why use both SP and FP?

# Variable Length Frame Size

- C allows allocating objects of unbounded size in the stack

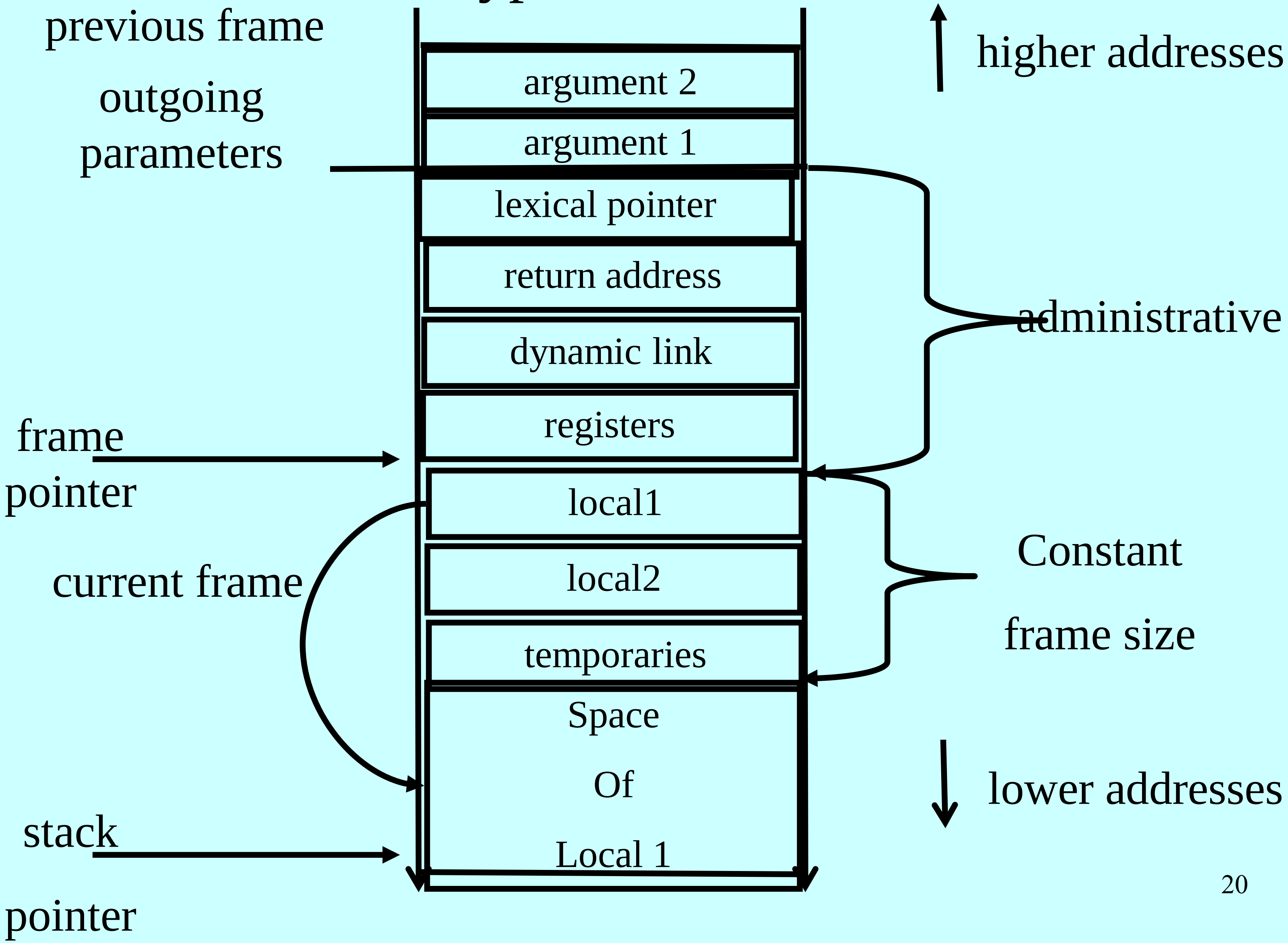
```
void p() {
   int i;
   char *p;
   scanf("%d", &i);
   p = (char *) alloca(i*sizeof(int));
}
```

- Some versions of Pascal allows conformant array value parameters

# Pascal Conformant Arrays

```
program foo ;
const max = 4 ;
var m1, m2, m3: array [1..max, 1..max] of  integer
var i, j: integer
procedure mult(a, b: array [1..l, 1..l]  of integer;
                    var c:array [1..l, 1..l] of integer));
   var i, j, k:  integer;
   begin { mult }
      for i := 1  to l do
         for j := 1 to l do begin
            c[i, j] := 0  ;
            for k := 1 to l do
               c[i, j] := c[i, j] + a[i, k] * b[k, j];
                              end
   end; { mult}
begin { foo}
    …
    mult(m1, m2, m3)
end. { foo}
```

# A Typical Stack Frame



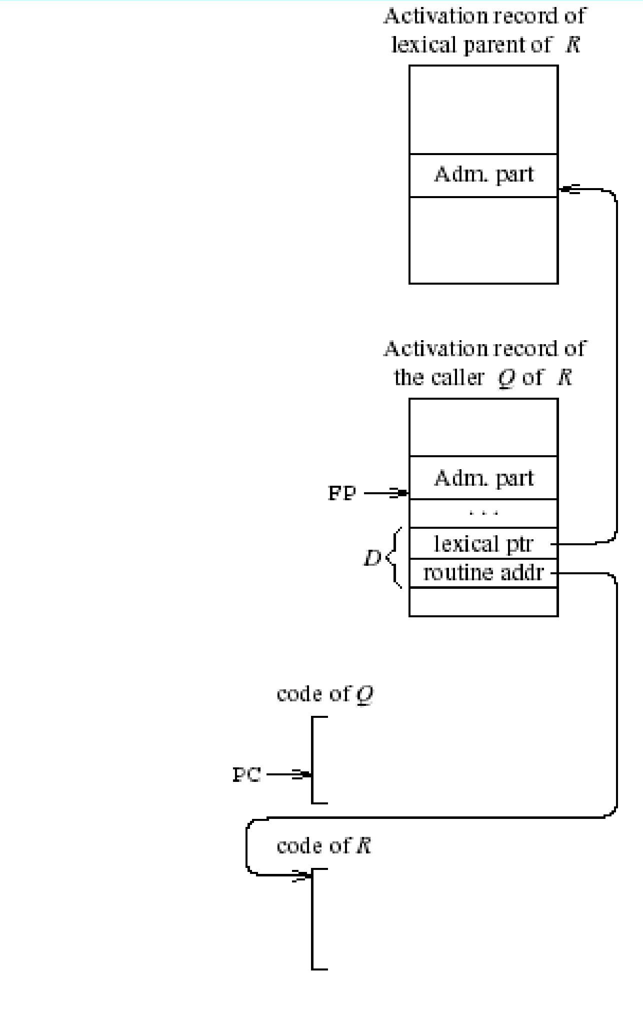
# Supporting Static Scoping

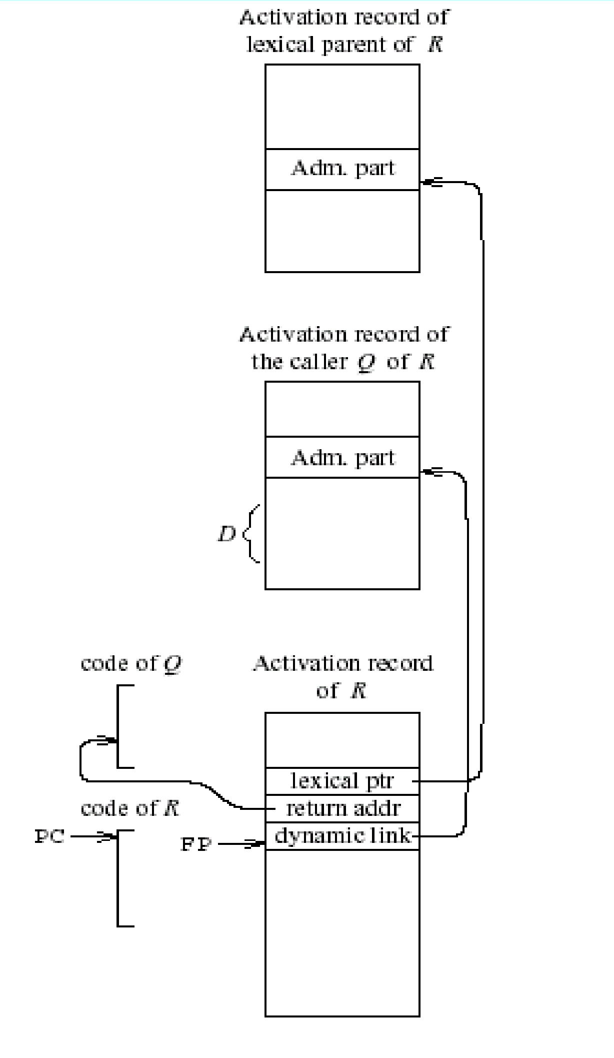
- References to non-local variables
- Language rules
  - No nesting of functions
    - C, C++, Java
  - Non-local references are bounded to the most recently enclosed declared procedure and “die” when the procedure end
    - Algol, Pascal, Scheme
- Simplest implementation
    - Pass the lexical pointer as an extra argument to functions
      - Scope rules guarantee that this can be done
    - Generate code to traverse the frames

# Routine Descriptor for Languages with nested scopes

| |
|---|
| Lexical pointer |
| routine address |

# Calling Routine R from Q



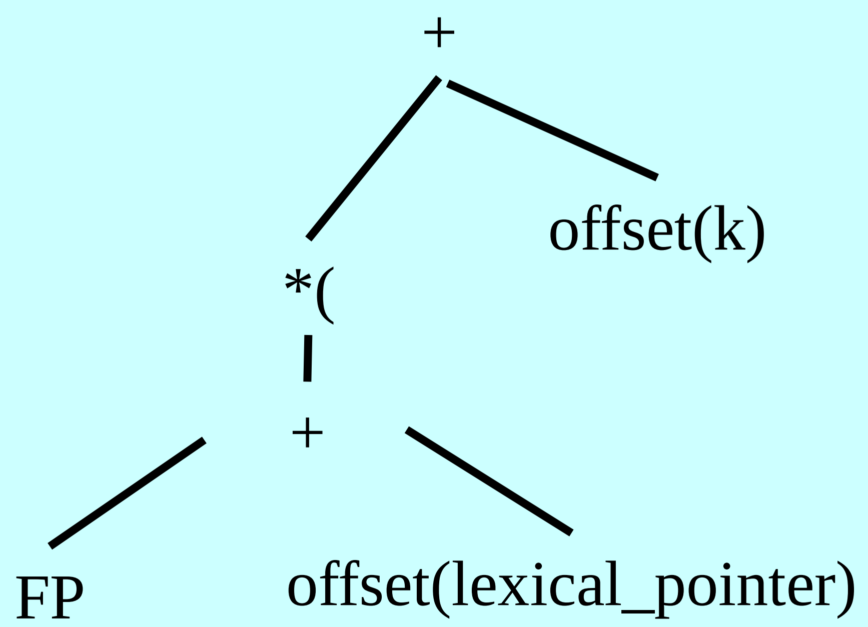



# Nesting Depth

- The semantic analysis identifies the static nesting hierarchy
- A possible implementation
  - Assign integers to functions and variables
  - Defined inductively
    - The main is at level 0
    - Updated when new function begins/ends

# Calculating L-Values

```
 0
int i;
void level_0(void){      1
   int j;
   void level_1(void) {      2
     int k;
     void level_2(void) {      3
        int l;
        k=l;
        j =l;
}}}
```



# Code for the k=l

```
int i;
void level_0(void){
   int j;
   void level_1(void) {
      int k;
      void level_2(void) {
         int l;
         k=l;
         j =l;
}}}
```

```
*(
    *(
        FP
        +
        offset(lexical_pointer)
    )
    +
    offset(k)
) =
*(FP + offset(l))
```

# Code for the j=l

```
int i;
void level_0(void){
   int j;
   void level_1(void) {
      int k;
      void level_2(void) {
         int l;
         k=l;
         j =l;
 }}}
```

```
*(
    *(
        *(  FP
            +
            offset(lexical_pointer)
        )
        +
        offset(lexical_pointer)
    )
    +
    offset(j)
) =
*(FP + offset(l))
```

# Other Implementations of Static Scoping

- Display
  - An array of lexical pointers
  - d[i] is lexical pointer nesting level i
  - Can be stored in the stack
- lambda-lifting
  - Pass non-local variables as extra parameters

# Machine Registers

- Every year
  - CPUs are improving by 50%-60%
  - Main memory speed is improving by 10%
- Machine registers allow efficient accesses
  - Utilized by the compiler
- Other memory units exist
  - Cache

# RISC vs. CISC Machines

| Feature | RISC | CISC |
|---|---|---|
| Registers | ⩾32 | 6, 8, 16 |
| Register Classes | One | Some |
| Arithmetic Operands | Registers | Memory+Registers |
| Instructions | 3-addr | 2-addr |
| Addressing Modes | r<br>M[r+c] (l,s) | several |
| Instruction Length | 32 bits | Variable |
| Side-effects | None | Some |
| Instruction-Cost | "Uniform" | Varied |

# Caller-Save and Callee-Save Registers

- Callee-Save Registers
  - Saved by the callee before modification
  - Values are automatically preserved across calls
- Caller-Save Registers
  - Saved (if needed) by the caller before calls
  - Values are not automatically preserved across calls
- Usually the architecture defines caller-save and callee-save registers
- Separate compilation
- Interoperability between code produced by different compilers/languages
- But compiler writers decide when to use calller/callee registers

# Callee-Save Registers

- Saved by the callee before modification
- Usually at procedure prolog
- Restored at procedure epilog
- Hardware support may be available
- Values are automatically preserved across calls

```
int foo(int a)    {
        int b=a+1;
        f1();
        g1(b);
        return(b+2);
}
```

```
.global _foo
    Add_Constant -K, SP //allocate space for foo
    Store_Local  R5, -14(FP) // save R5
    Load_Reg  R5, R0; Add_Constant R5, 1
    JSR f1 ; JSR g1;
    Add_Constant R5, 2; Load_Reg R5, R0
    Load_Local -14(FP), R5 // restore R5
    Add_Constant K, SP; RTS // deallocate
```

# Caller-Save Registers

- Saved by the caller before calls when needed
- Values are not automatically preserved across calls

```
void bar (int y) {
        int x=y+1;
        f2(x);
    g2(2);
    g2(8);
}
```

```
.global _bar
        Add_Constant -K, SP //allocate space for bar
        Add_Constant R0, 1
        JSR f2
        Load_Constant  2, R0  ;       JSR g2;
        Load_Constant 8, R0 ;         JSR g2
        Add_Constant K, SP // deallocate space for bar
        RTS
```

# Parameter Passing

- 1960s
  - In memory
    - No recursion is allowed
- 1970s
  - In stack
- 1980s
  - In registers
  - First k parameters are passed in registers (k=4 or k=6)
  - Where is time saved?

- Most procedures are leaf procedures
- Interprocedural register allocation
- Many of the registers may be dead before another invocation
- Register windows are allocated in some architectures per call (e.g., sun Sparc)

# Modern Architectures

- return-address
  - also normally saved in a register on a call
  - a non leaf procedure saves this value on the stack
  - No stack support in the hardware
- function-result
  - Normally saved in a register on a call
  - A non leaf procedure saves this value on the stack

# Limitations

- The compiler may be forced to store a value on a stack instead of registers
- The stack may not suffice to handle some language features

# Frame-Resident Variables

- A variable x cannot be stored in register when:
  - x is passed by reference
  - Address of x is taken (&x)
  - is addressed via pointer arithmetic on the stack-frame (C varags)
  - x is accessed from a nested procedure
  - The value is too big to fit into a single register
  - The variable is an array
  - The register of x is needed for other purposes
  - Too many local variables
- An escape variable:
  - Passed by reference
  - Address is taken
  - Addressed via pointer arithmetic on the stack-frame
  - Accessed from a nested procedure

# The Frames in Different Architectures

g(x, y, z) where x escapes

| | Pentium | MIPS | Sparc |
|---|---|---|---|
| x | InFrame(8) | InFrame(0) | InFrame(68) |
| y | InFrame(12) | InReg($X_{157}$) | InReg($X_{157}$) |
| z | InFrame(16) | InReg($X_{158}$) | InReg($X_{158}$) |
| View Change | M[sp+0]$\leftarrow$fp<br>fp $\leftarrow$sp<br>sp $\leftarrow$sp-K | sp $\leftarrow$sp-K<br>M[sp+K+0] $\leftarrow r_2$<br>$X_{157} \leftarrow$r4<br>$X_{158} \leftarrow$r5 | save %sp, -K, %sp<br>M[fp+68]$\leftarrow i_0$<br>$X_{157} \leftarrow i_1$<br>$X_{158} \leftarrow i_2$ |

# The Need for Register Copies

```
void m(int x, int y) {
   h(y, y);
   h(x, x);
}
```

# Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P
- Example 1: Static variables in C
  (own variables in Algol)
```
void p(int x)
{
  static int y = 6 ;
  y += x;
 }
```
- Example 2: Features of the C language
```
int * f()
{ int x ;
  return &x ;
}
```
- Example 3: Dynamic allocation
```
int * f()  { return (int *) malloc(sizeof(int)); }
```

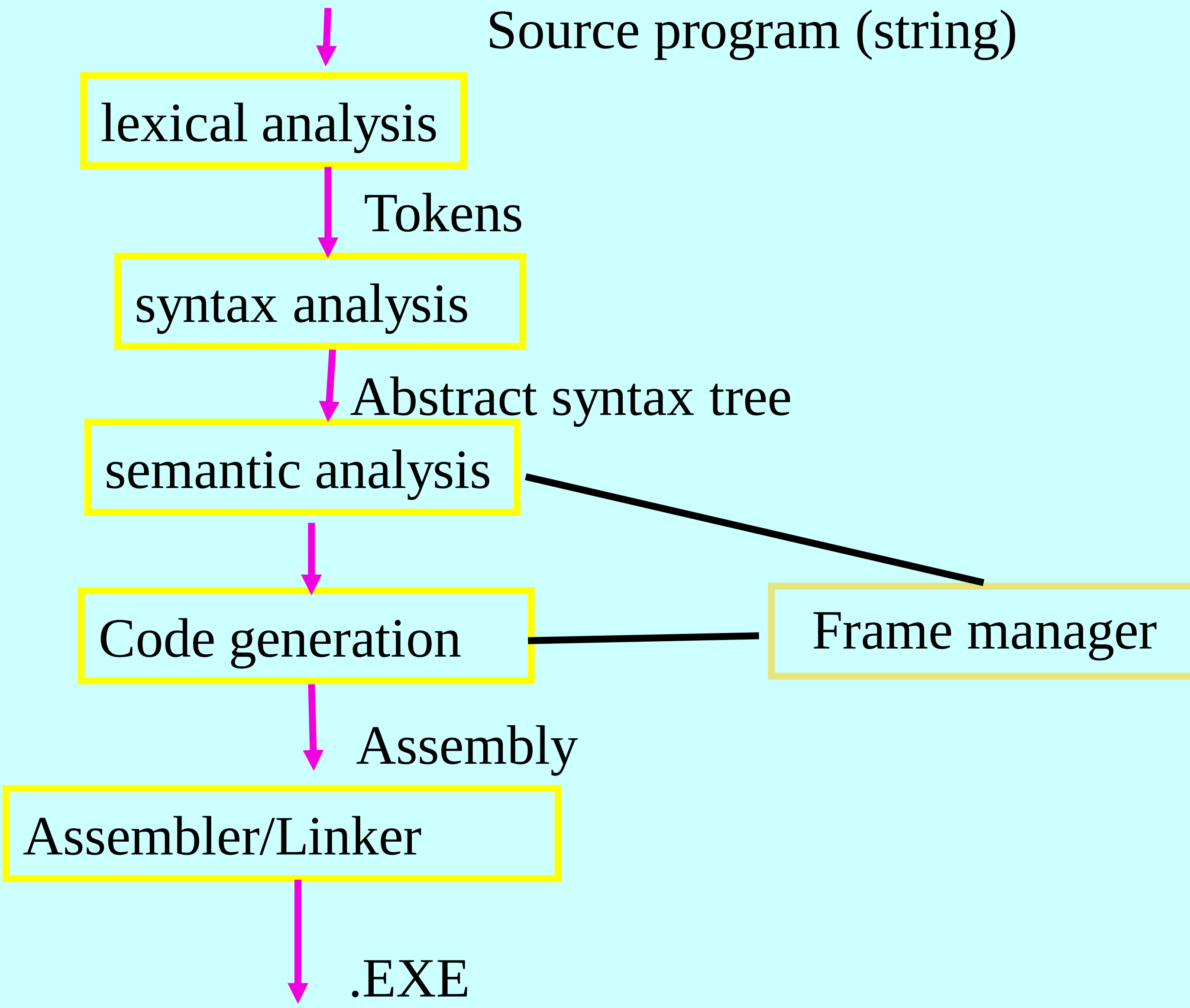
# Currying Functions

```
int (*)() f(int x)
{
   int g(int y)
   {
       return x + y;
    }
    return g ;
 }

int (*h)() = f(3);
int (*j)()  = f(4);

int z = h(5);
int  w = j(7);
```

# Compiler Implementation

- Hide machine dependent parts
- Hide language dependent part
- Use special modules

# Basic Compiler Phases



# Hidden in the frame ADT

- Word size
- The location of the formals
- Frame resident variables
- Machine instructions to implement “shift-of-view” (prologue/epilogue)
- The number of locals “allocated” so far
- The label in which the machine code starts

# Invocations to Frame

- “Allocate” a new frame
- “Allocate” new local variable
- Return the L-value of local variable
- Generate code for procedure invocation
- Generate prologue/epilogue
- Generate code for procedure return

# Summary

- Stack frames provide a simple compile-time memory management scheme
  - Locality of references is supported
- Can be complex to implement
- Limits the duration of allocated objects
- Memory allocation is one of most interesting areas