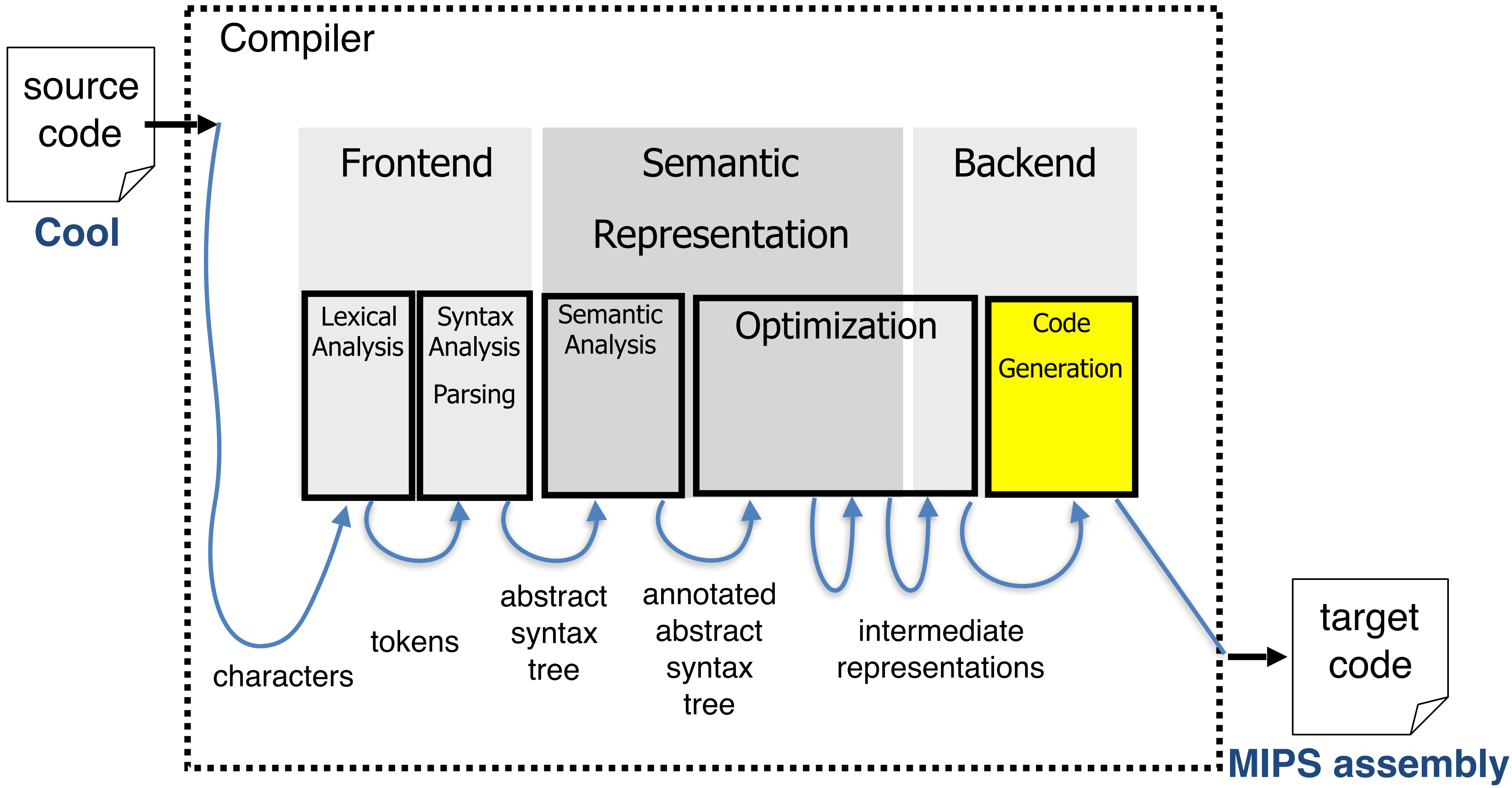
Runtime organization

STACK FRAMES
(ACTIVATION RECORDS)

Greta Yorsh


source code
Cool
Compiler
Frontend
Semantic Representation
Backend
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
Code Generation
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code
MIPS assembly

Supporting function calls

- How is that done?
- What do we need from the compiler?

n = f(a[i])

```
t1 = i * 4
t2 = a + t1
param t2
t3 = call f, 1
n = t3
```

Supporting function calls

- ✓ Type checking
  - function type: return type, type of formal parameters
  - within an expression function treated like any other operator
- ✓ Symbol table
  - parameter names
- New computing environment (scope)
  - at least temporary memory for local variables
- Pass information into the new environment
  - parameters
- Transfer of control to/from method and handle return values

Design decisions

- Scoping rules
  - static scoping vs. dynamic scoping
- Caller/callee conventions
  - parameters
  - who saves register values?
- Allocating space for local variables

Static scoping

```
main ( )
{
      int a = 0 ;
      int b = 0 ;
      {
          int b = 1 ;
          {
              int a = 2 ;
              printf ("%d %d\n", a, b);
          }
          {
              int b = 3 ;
              printf ("%d %d\n", a, b) ;
          }
          printf ("%d %d\n", a, b) ;
      }
      printf ("%d %d\n", a, b) ;
}
```

a name refers to its (closest) enclosing scope

known at **compile** time

Output:

```
2 1
0 3
0 1
0 0
```

Static scoping

```
main ( )
{
        int a = 0 ;
        int b = 0 ;
        {
            int b = 1 ;
            {
                int a = 2 ;
                printf ("%d %d\n", a, b);
            }
            {
                int b = 3 ;
                printf ("%d %d\n", a, b) ;
            }
            printf ("%d %d\n", a, b) ;
        }
        printf ("%d %d\n", a, b) ;
}
```

B_0 B_1 B_2 B_3

a name refers to its (closest) enclosing scope

known at **compile** time

Declaration	Scopes
a=0	B0,B1,B3
b=0	B0
b=1	B1,B2
a=2	B2
b=3	B3

Dynamic scoping

- Scope determined at runtime
- Each identifier is associated with a global stack of bindings
- When entering scope where identifier is declared
  - push declaration on identifier stack
- When exiting scope where identifier is declared
  - pop identifier stack
- **Evaluating the identifier in any context binds to the current top of stack**

Example

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

- What value is returned from main?
  - static scoping
  - dynamic scoping

Example: static vs dynamic scope

```
int x = 37;
int y = 42;
void Function1() {
    Print(x + y);
}
void Function2() {
    int x = 0;
    Function1();
}
void Function3() {
    int y = 0;
    Function2();
}
Function1();
Function2();
Function1();
Function3();
```

static scoping output:
79
79
79
79

dynamic scoping output:
79
42
79
0

Why do we care?

- We need to generate code to access variables

- Static scoping
  - identifier binding is known at compile time
  - address of the variable is known at compile time
  - assigning addresses to variables is part of code generation
  - no runtime errors of “access to undefined variable”
  - can check types of variables

Variable addresses for static scoping first attempt

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { return g(); }
```

identifier	address
x (global)	0x24
x (inside g)	0x72

Variable addresses for static scoping first attempt

```
int a [11] ;

void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quicksort (m, i-1) ;
    quicksort (i+1, n) ;
  }

main() {
  ...
  quicksort (1, 9) ;
}
```

What is the address of variable “i” in quicksort?

- How do we handle recursion?

Runtime stack

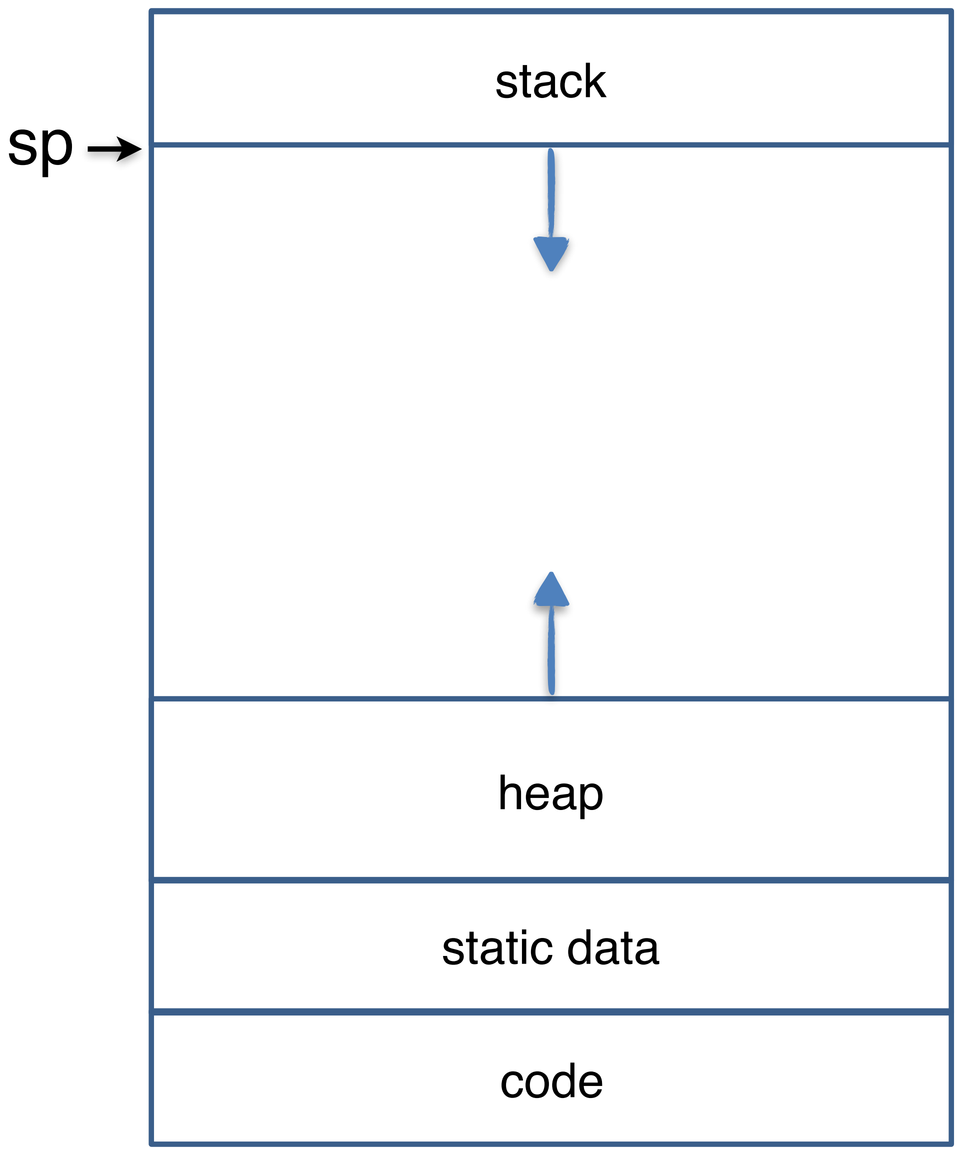
- Stack of frames

- Call: push new frame
- Return: pop frame
- Only one “active” frame: top of stack

Stack frame

- Separate space for **each function invocation**
- Variable size
  - different functions may require different memory sizes
  - size may be input dependent
- Managed **at runtime**
  - allocated upon function call
  - deallocated upon function return
  - efficient! function are called frequently
- Code for managing it generated by the compiler

Memory layout

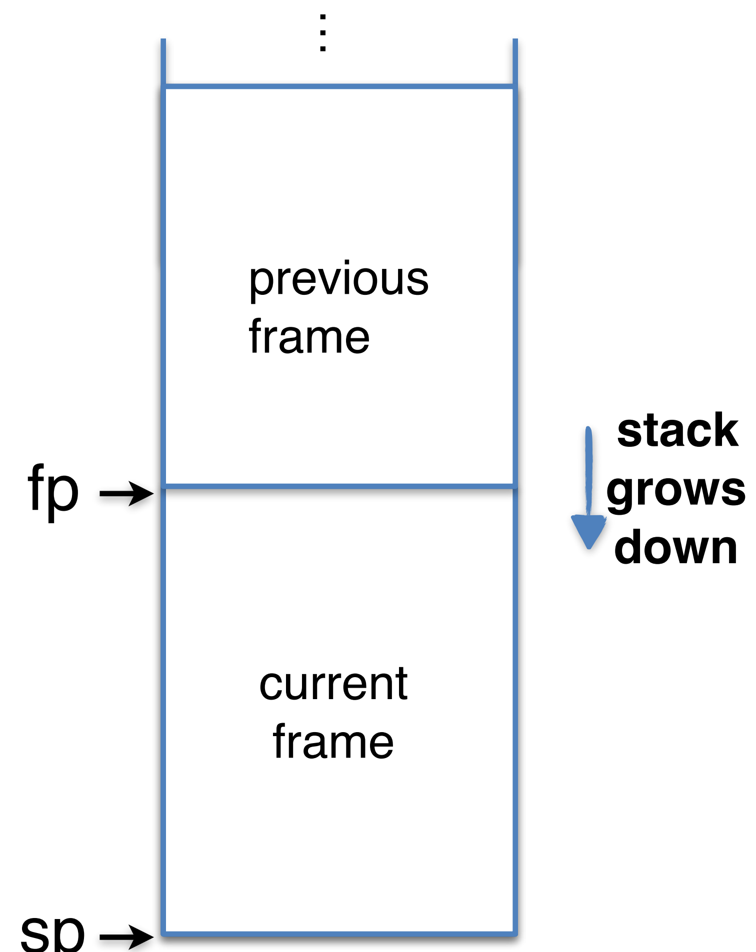


stack grows down
(towards lower addresses)

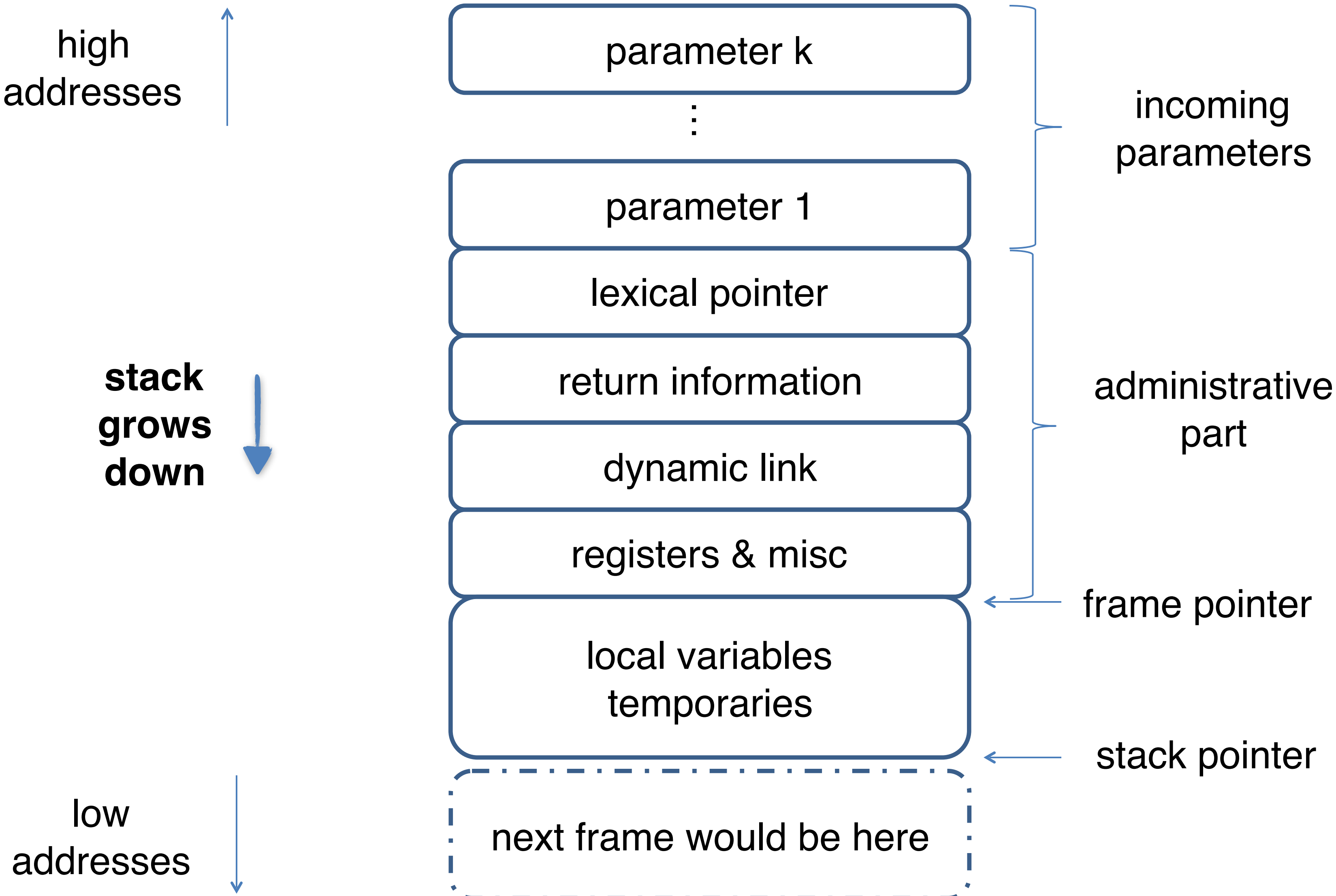
heap grows up
(towards higher addresses)

Runtime stack

- SP is **stack pointer**
  - top of current frame
- FP is **frame pointer**
  - base of current frame
  - sometimes called BP (base pointer)



Stack frame

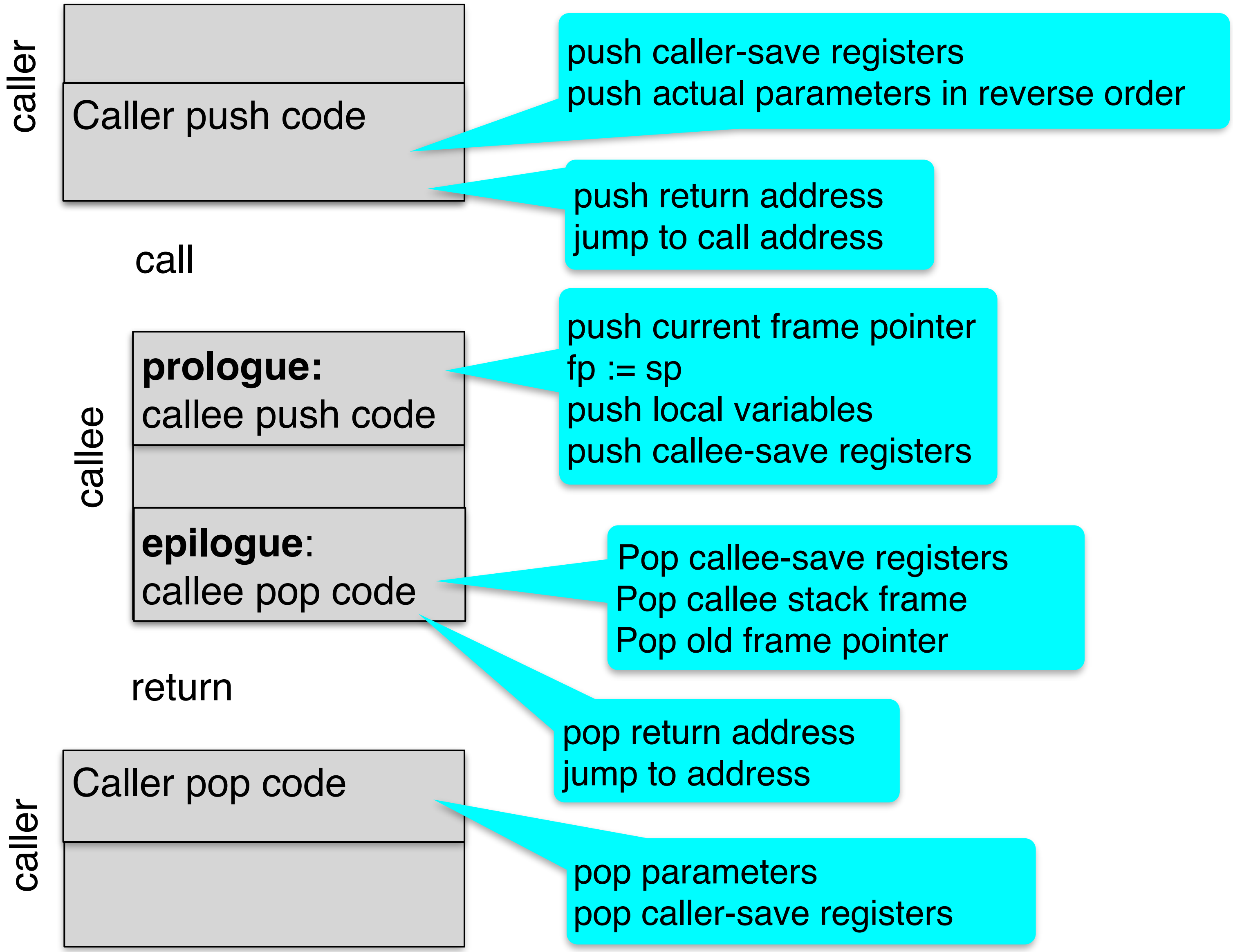

high addresses
stack grows down
low addresses
parameter k
⋮
parameter 1
lexical pointer
return information
dynamic link
registers & misc
local variables
temporaries
next frame would be here
incoming parameters
administrative part
frame pointer
stack pointer

Call sequence

- **The processor does not save the content of registers on function calls**

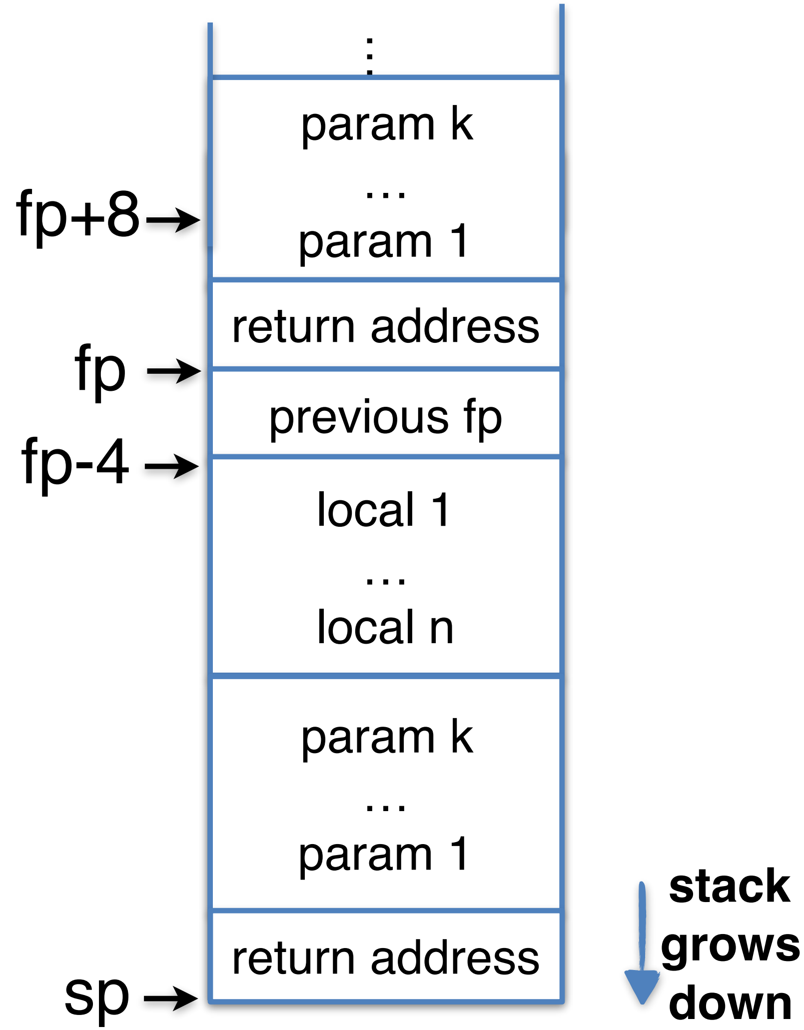
- So who will?
    - caller saves and restores registers
    - callee saves and restores registers
    - but can also have each save/restore some registers

Call sequence



Accessing stack variables

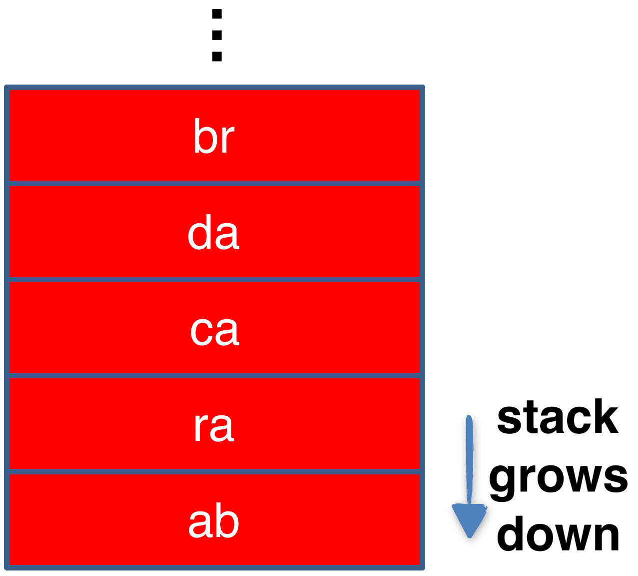
- Use offset from fp
- Remember:
  stack grows downwards
- Above fp = parameters
- Below fp = locals
- Examples
  - fp + 4 = return address
  - fp + 8 = first parameter
  - fp – 4 = first local



Buffer overflow

```
void foo (char *x) {
  char buf[2];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

gcc bf
./a.out abracadabra
Segmentation fault



Buffer overflow attack

```
int check_authentication(char *password) {

int auth_flag = 0;
char password_buffer[16];

  strcpy(password_buffer, password);
  if(strcmp(password_buffer, "brillig") == 0)
   auth_flag = 1;
  if(strcmp(password_buffer, "outgrabe") == 0)
   auth_flag = 1;
  return auth_flag;
}
int main(int argc, char *argv[]) {  if(argc < 2) {
   printf("Usage: %s <password>\n", argv[0]); exit(0);  }
   if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");    }
     else {
         printf("\nAccess Denied.\n");
     }
}
```

(source: “hacking – the art of exploitation, 2nd Ed”)

Stack frames

- Allocate a separate space for every function invocation
- Naturally supports recursion
- Efficient memory allocation policy
- Provides a simple way to achieve modularity

CALLING CONVENTIONS

Calling conventions: who and why?

- Microprocessor manufacturers specify “standard” schemes to be used by all compilers
  - stack layout
  - registers for parameter passing and return values
  - callee vs caller saved registers
- Functions compiled with one compiler can call functions compiled with another
- Essential for interaction with libraries and runtime system

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)

Modern computer architectures

- Parameters
  - first k parameters are passed in registers
  - others on the stack
- Return address
  - automatically saved in a register on a call
  - a non-leaf function saves this value on the stack
- Function result
  - normally saved in a register on a return
- No stack support in the hardware (why?)

Stack Operations in RISC

- PUSH
  - sub $sp, 4
  - sw $ra, ($sp)
- POP
  - lw $ra, ($sp)
  - add $sp, 4
- TOP
  - lw $ra, ($sp)

MIPS instructions for calls

Instruction	Meaning
jal my_proc	jump and link start procedure my_proc $ra holds address of instruction following the jal
jr $ra	jump register return from procedure call puts $ra value back into the PC

MIPS Calling Convention

- Caller
  - pass arguments: first 4 are in $a0-$a3 the rest pushed on the stack
  - save caller-saved registers, including $t0-$t9 if needed
  - jal ($ra gets address of instruction following the jal)
- Callee
  - save callee-saved registers in the frame
    - $fp
    - $ra (if the callee is not a leaf)
    - $s0-$s7 (if used by the callee)
  - push a stack frame $fp := $sp
  - …… do some work
  - return value is in $v0
  - restore callee-saved registers
  - pop the stack frame $sp := $fp
  - jr $ra (puts $ra value back into the program counter $pc)

Factorial Example

```c
int factorial (int n){
  if (n < 1) return 1;
  return (n * factorial (n-1));
}
```

```
factorial:
       bgtz  $a0  doit
       li    $v0  1          # base case, 0! = 1
       jr    $ra
doit:
       addiu $sp $sp -8       # stack frame
       sw    $s0 0($sp)       # will use for argument n
       sw    $ra 4($sp)       # return address
       move  $s0 $a0          # save argument
       addiu $a0 $a0  -1      # n-1
       jal   factorial        # v0 := (n-1)!
       mul   $v0 $s0 $v0      # n*(n-1)!
       lw    $s0 0($sp)       # restore registers from stack
       lw    $ra 4($sp)
       addiu $sp $sp 8
       jr    $ra
```

To callee-save or caller-save?

- Callee-saved registers will contain the same value before and after the call
- Callee-saved registers need only be saved when callee modifies their value
- Caller-saved registers need only be saved when their value is used after call
- Callee can use caller-save registers without saving
- Caller need not save callee-saved registers before a call

- Placement of values into callee-save vs. caller-save registers determined by the **register allocator**
- Some heuristics and conventions are followed

To callee-save or caller-save?

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

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

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
  - Sun’s Sparc

RUNTIME STACK: ADVANCED TOPICS

Nested procedures

- Pascal
  - any routine can have sub-routines
  - any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
  - “non-local” variables
- Caml, Scala, Javascript, C#, Java8,…
  - lambda expressions, closures
- Are frames allocated on stack or heap?

Example: nested procedures

```
void p()
{
    int x;
    void a()
    {
        int y;
        void b () {  … c() … };
        void c ()
        {
            int z;
            void d () {  y = x + z  };
            … b() … d() …
        }
        … a() … c() …
    }
    a()
}
```

B_0 B_1 B_2

possible call sequence:
p→a→a→c→b→c→d

what is the address of variable "y" in procedure d?

Nested procedures

- Procedures may need to access variables of another procedure that contains it
- How do you find the right stack frame at runtime?

- When **c** uses variables from **a**, which instance of **a** to use?

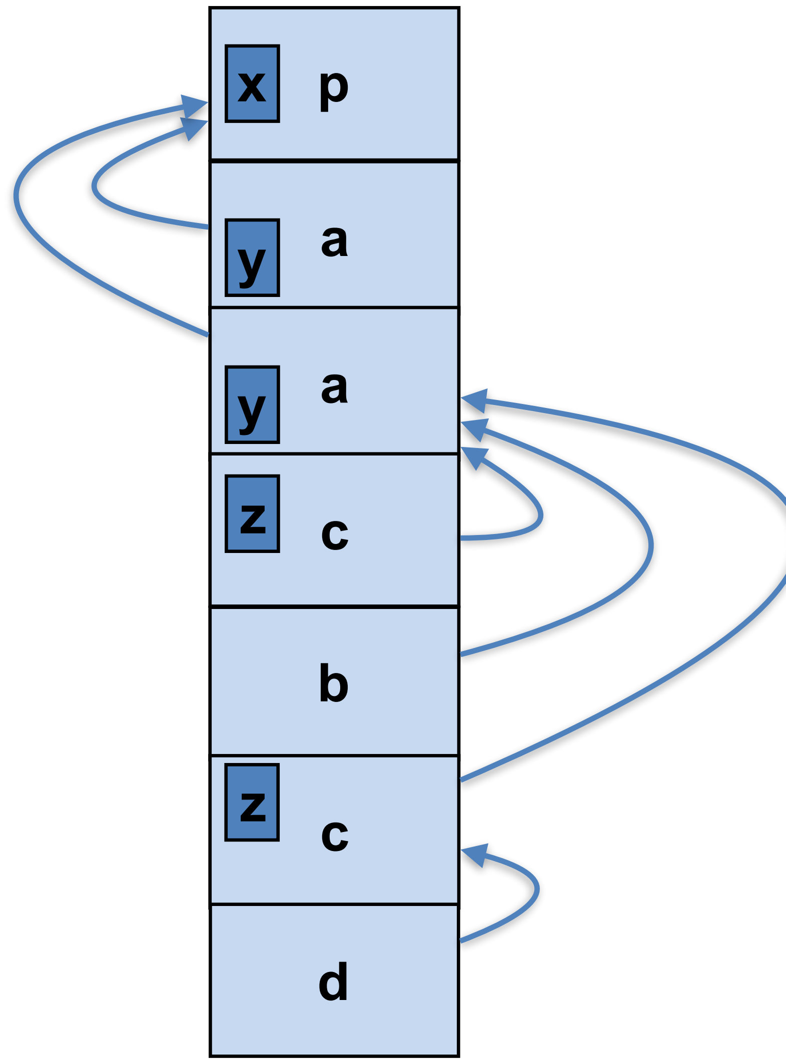
Lexical pointer

- Points to the **last frame** of the (static) nesting level above it
- Created at runtime and stored in the stack frame
- Lexical pointer of **c** points to a stack frame of **a**

- Accessing a variable requires a chain of indirect memory references through lexical pointers
- Number of links in the chain is
  - the difference in nesting depth between declaration scope and use scope of the variable
  - known at compile time

Example: nested procedures

```
       void p()
       {
           int x;
           void a()
           {
                 int y;
                 void b () {  … c () … };
B0    B1         void c ()
                 {
          B2           int z;
                       void d () {  y = x + z  };
                     … b() … d() …
                  }
                 … a() … c() …
            }
            a()
       }
```



possible call sequence:

p→a→a→c→b→c→d

Terminology

- **old frame pointer** = dynamic link = control link points to the stack frame of the caller

- **lexical pointer** = static link = access link points to the last frame of the closest lexically enclosing block in program text

Is lexical pointer needed in C?

Old frame pointer

- **Optional** if size of caller's stack frame is known at compile time
- **Convenient** in many situations:
  - fixed offsets
    - positive for locals
    - negative for parameters
  - quickly restore stack pointer during return sequence
  - dynamic allocation on stack: alloca(…) in C/C++
  - variable length parameter lists: vararg in C/C++

Tail call elimination

- Tail call is a call performed as **the last operation** of the caller
- Optimization: set return address to that of caller
- Can we do the same with old frame pointer?
- Can we reuse the entire frame of the caller?
- Does it prevent stack overflow?
- Does it work with dynamic scope?

```
void f0 () {
  f1(1);
  f2(2);
}
```

```
int f1 (int x) {
  int y;
  if (x > 0)
    return f2(x-1);
  else
    return f2(x);
}
```

```
int f2 (in y) {
  return y+42;
}
```

```
int f3 (int z) {
  return f2(z)+42;
}
```

Tail recursion

- Tail recursive function is equivalent to a loop
- Example: compute the least power of 2 greater than y

```
int g(int y) { g1(1,y); }
int g1(int x, int y) {
  if (x>y)
    return x;
  else
    return g1(2*x, y);
}
```

```
int g(int y) {
  int x = 1;
  while !(x>y)
    x = 2*x;
  return x;
}
```

Address of formal parameters

- If the address of a formal parameter is taken, callee writes register to stack
- Cannot save formal parameter in the temporary area of the frame, with other callee saved registers, because all parameters must be consecutive in memory
- Space for the parameter must be allocated on the stack, even if the parameter is passed in a register
- Example: C/C++

```
void foo(int x) {
   bar(&x);
   return x + 1;
}
```

```
void bar(int *a) {
   *a = *a + 2;
}
```

Higher-order functions

- Functions passed as argument
- Need pointer to a frame “higher up” in stack

- Function returned as the result of function call
- Local variables of a function need to remain even after function returned

- Allocate frames on the heap, not stack