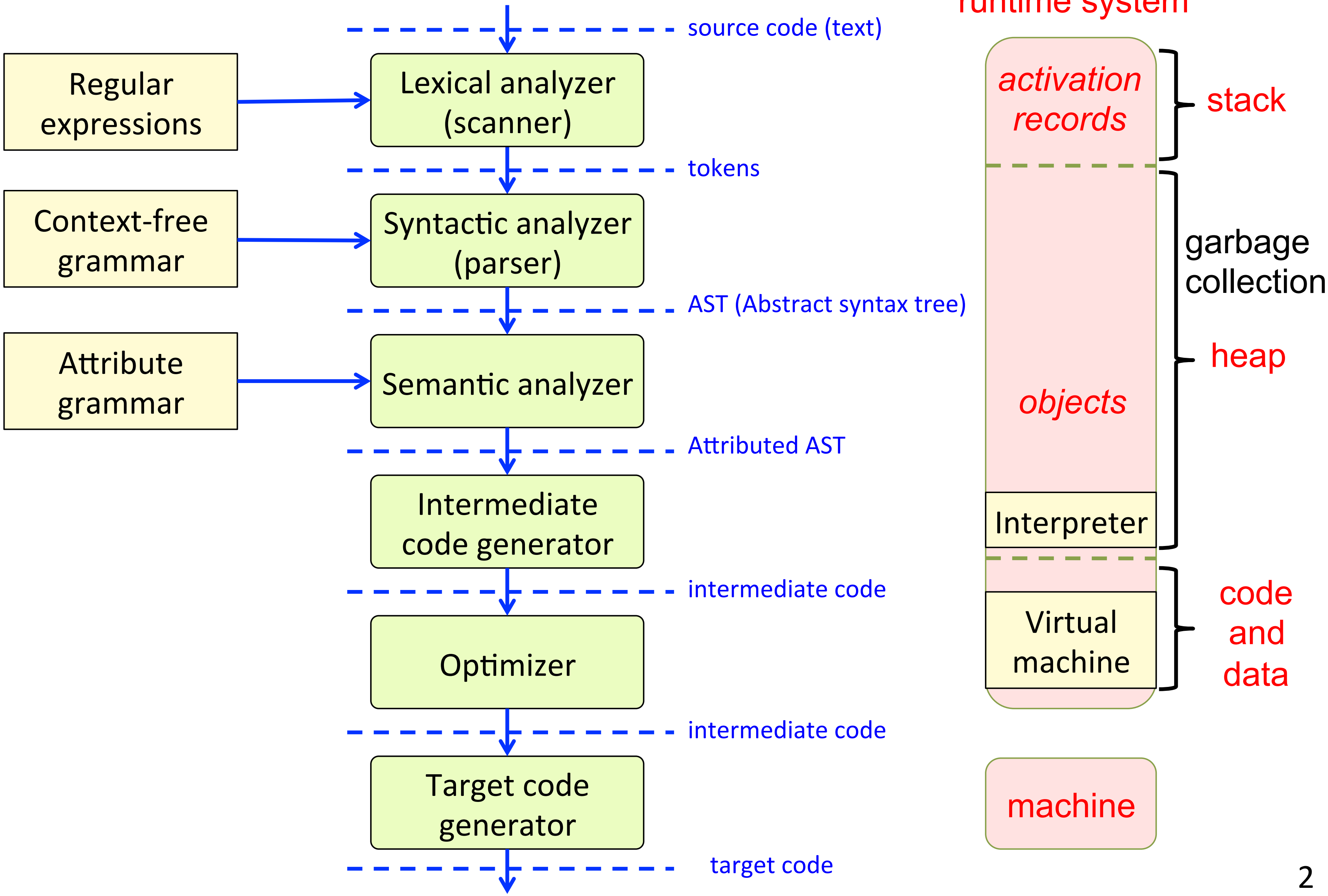
EDAN65: Compilers, Lecture 10

Runtime systems

Görel Hedin

Revised: 2015-09-22

This lecture



Runtime systems

Organization of data

- Global/static data
- Activations (method instances)
- Objects (class instances)

Method calls

- Call and return
- Parameter transmission

Access to variables

- Local variables
- Non-local variables

Object-oriented constructs

- Inheritance
- Overriding
- Dynamic dispatch
- Garbage collection

The machine

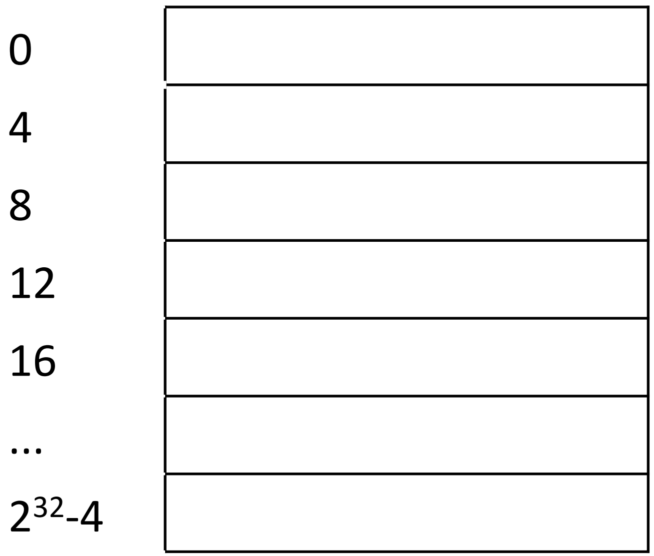
Registers: 32 or 64 bits wide

Typically a small number.
For example, 32 registers

Some have dedicated roles:
program counter, stack pointer, ...

Some are general purpose, for computations

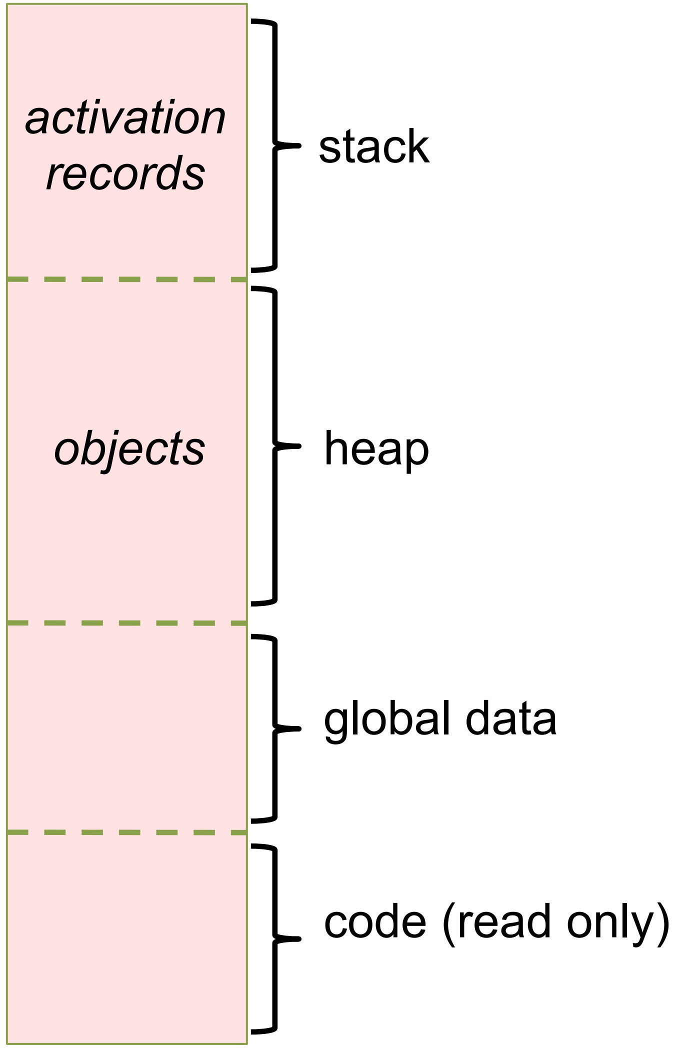
Memory: Typically byte adressed



Very large – order of Mbyte or Gbyte.
Like a very big array.

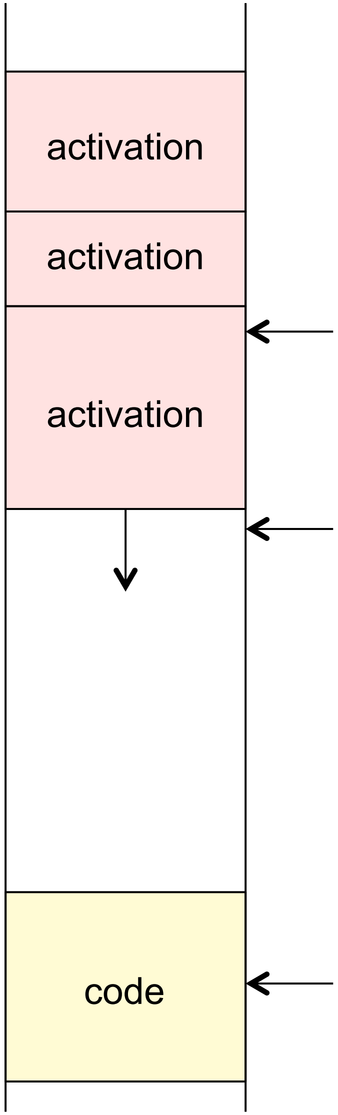
Typically divided into different segments:
global data, code, stack, heap.

Example memory segments



Stack of activations

The data for each method call is stored in an **activation**



stack grows

Some dedicated registers:

FP – Frame Pointer. The first word of the current activation

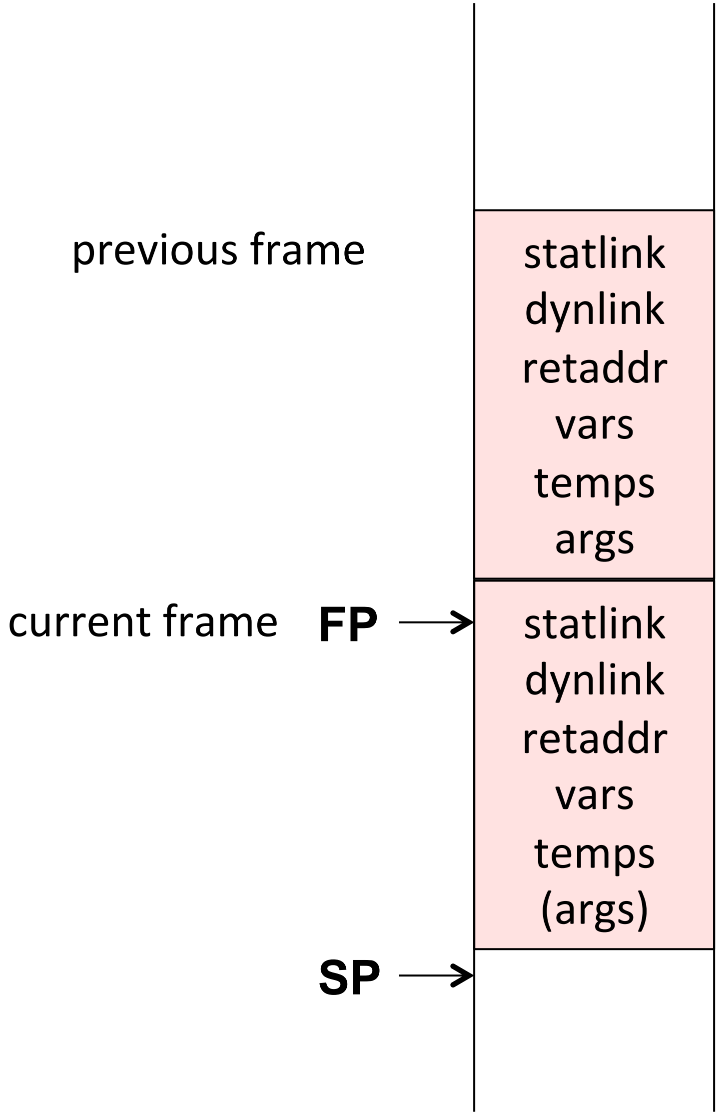
SP – Stack Pointer. The first unused word of the stack

PC – Program counter. The currently executing instruction.

Synonyms:
activation record
activation
stack frame
frame

Swedish:
aktiveringspost

Example activation layout



The calling method pushes arguments on the stack.
The return value is placed in a register.

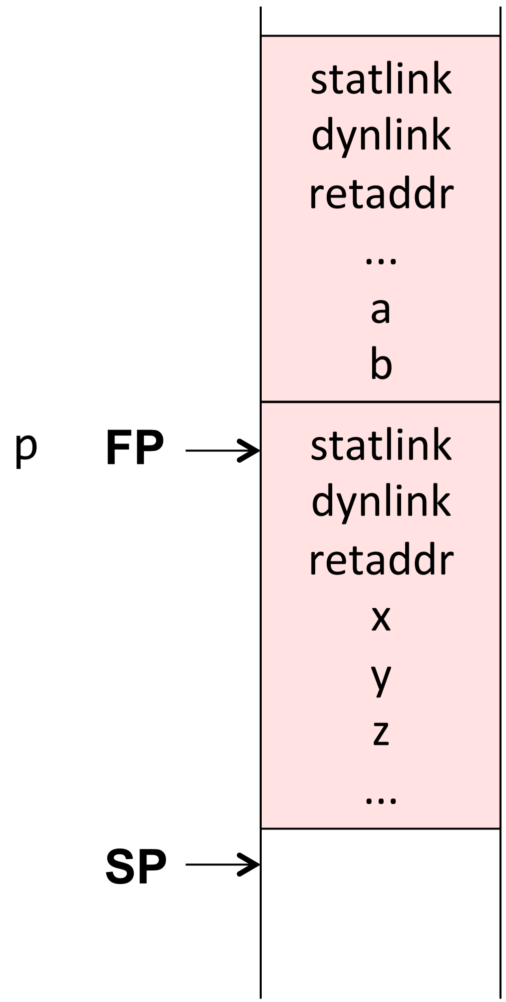
args: Arguments to the next called method.

statlink: Frame of enclosing method/object
dynlink: Frame of calling method
retaddr: Saved PC - where to jump at return
vars: Local variables
temps: Temporary variables and saved registers
args: Arguments to the next called method.

Frame pointer

Used for accessing arguments and variables in the frame

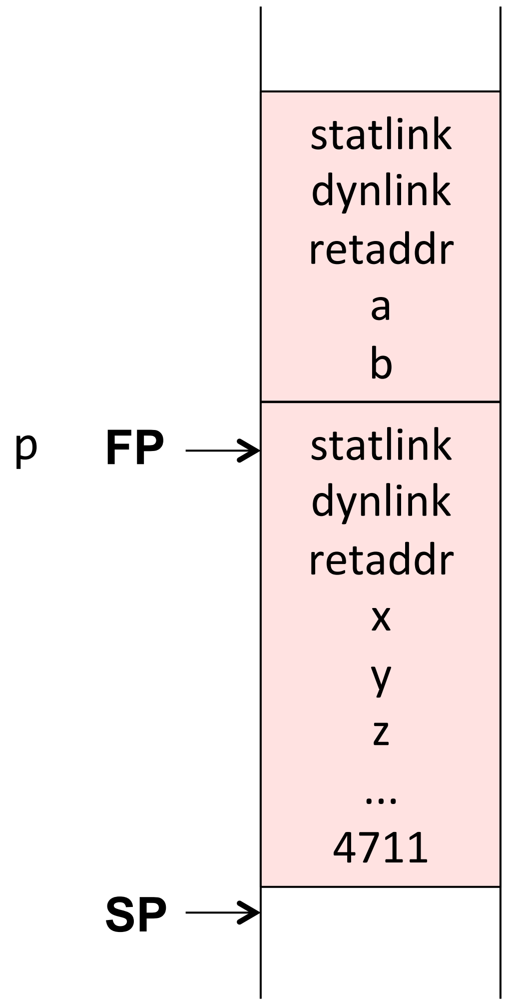
```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  ...
}
```



Stack pointer

Used for growing the stack, e.g., at a method call

```
void p(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  q(4711);
}
```

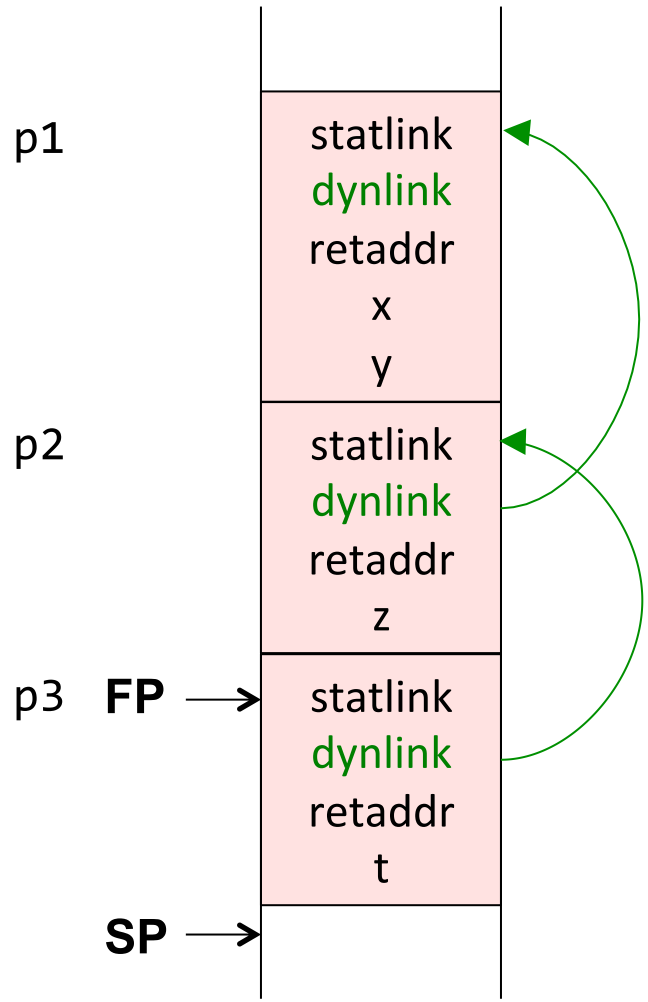


The argument 4711 is pushed on the stack before calling q

Dynamic link

Points to the frame of the calling method

```
void p1() {
  int x = 1;
  int y = 2;

  void p2() {
    int z = y+1;
    p3();
  }

  void p3(){
    int t = x+3;
  }

  p2(); y++;
}
```



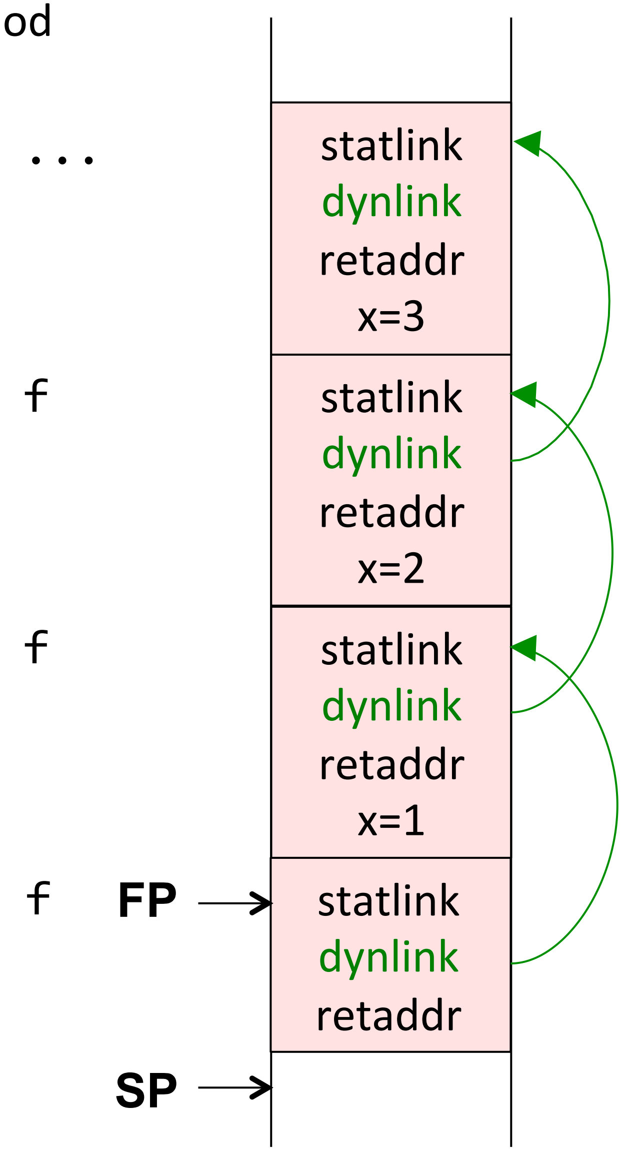
Used for restoring FP when returning from a call.

Recursion

Several activations of the same method

```
int f(int x) {
  if (x <= 1)
    return 1;
  else
    return x * f(x-1);
}
```

```
...
f(3);
...
```

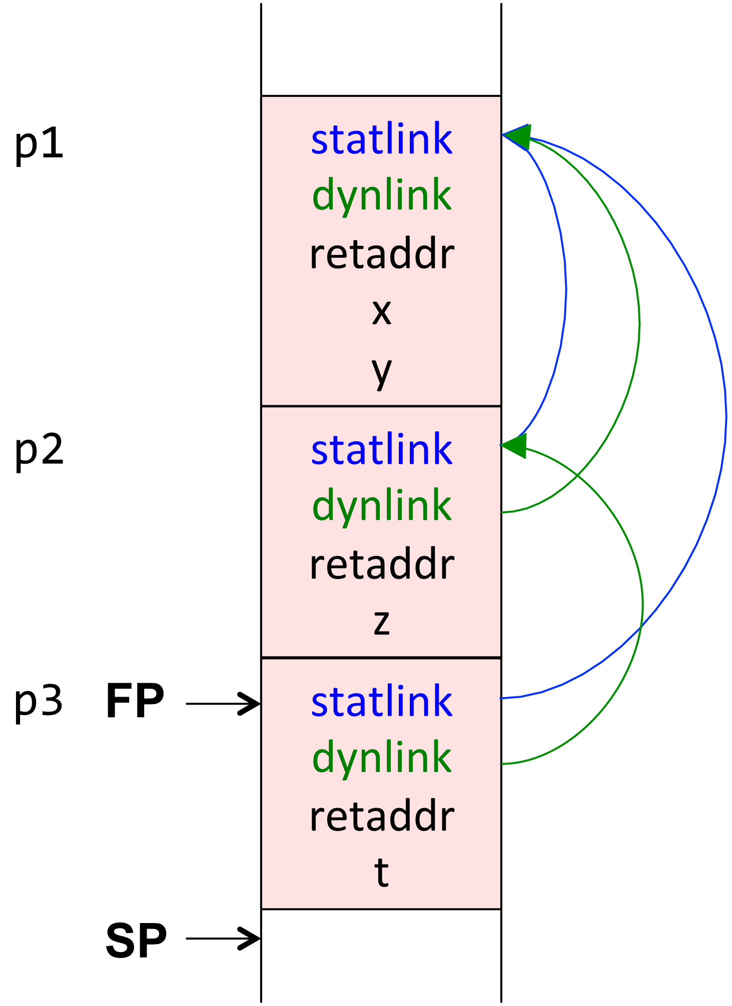


Static link

Points to the frame of the *enclosing* method.
Makes it possible to access variables in enclosing methods.

```
void p1() {
  int x = 1;
  int y = 2;

  void p2() {
    int z = y+1;
    p3();
  }

  void p3(){
    int t = x+3;
  }

  p2(); y++;
}
```

The methods are *nested.*
Supported in Algol, Pascal,
but not in C, Java...

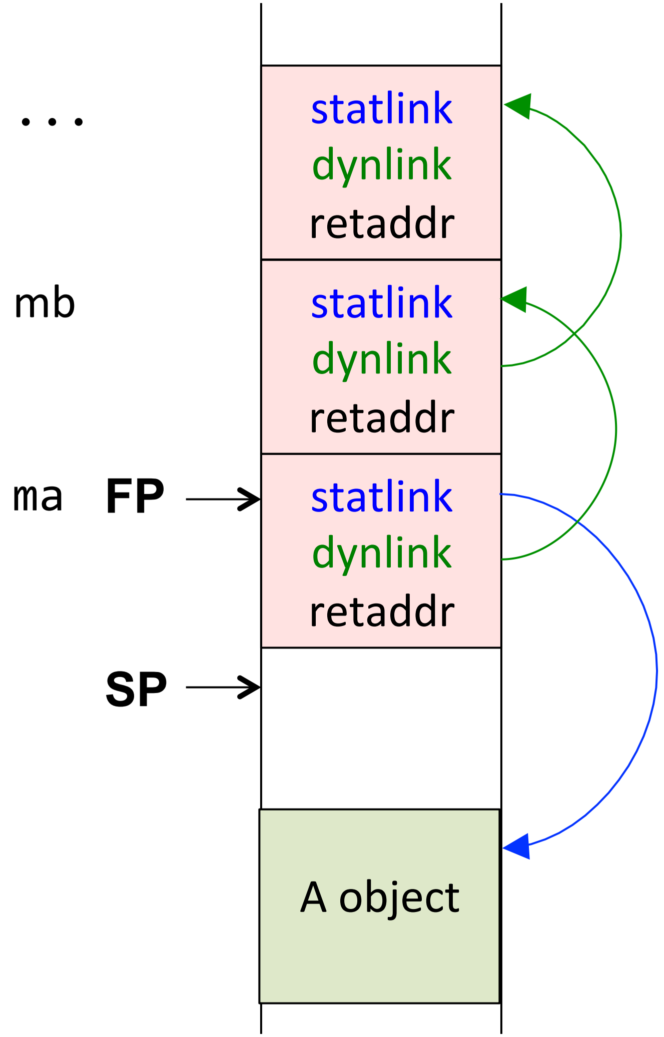


Static link in OO programs

Corresponds to the *this* pointer.
Makes it possible to access fields in the object.

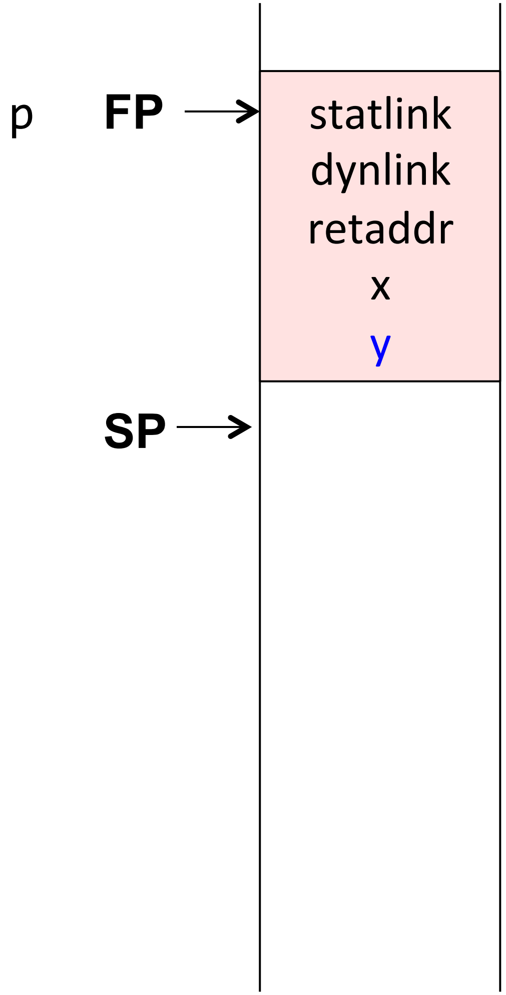
```
class A {
  int x = 1;
  int y = 2;

  void ma() {
    x = 3;
  }
}
```

```
class B {
  void mb() {
    A a = ...;
    a.ma();
  }
}
```



Access to local variable

```
void p() {
  int x = 1;
  int y = 2;
  y++;
  ...
}
```



Assume all variables can be stored in one 32 bit word.

The compiler enumerates the variables:
nr(x) = 0
nr(y) = 1

Compute offset relative FP
offset(y) = headersize + nr(y)*4 = 12+4 = 16

Typical assembly code for y++

```
ADD   FP #16 R1   // R1 is now the address to y
LOAD  R1 R2       // R2 is now the current value of y
INC   R2          // Increment R2
STORE R2 R1       // Store the new value to memory
```

Computing offsets for variables

```
void p() {
  boolean f1 = true;
  int x = 1;
  boolean f2 = false;
  if (...) {
    int y = 2;
    ...
  }
  else {
    int z = 3;
    ...
  }
  ...
}
```

The compiler can reorder variables in the activation to make efficient use of the space.

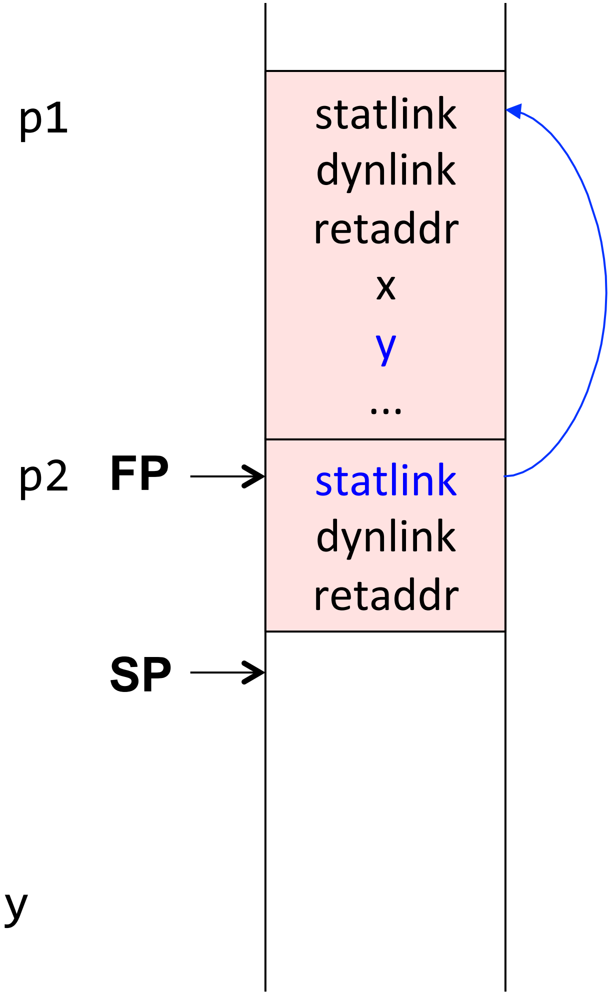
y and z have disjoint lifetimes.
They could share the same memory cell.

The booleans could be stored in consecutive bytes, or bits.

...

Access to non-local variable

```
void p1() {
  int x = 1;
  int y = 2;
  void p2() {
    y++;
  }
  p2();
}
```



The compiler knows that y is declared in the enclosing block.

Follow the static link once to get to the enclosing frame

```
LOAD  FP R1        // R1 is now the address to
                   // p1's frame

ADD   R1 #16 R2    // R2 is now the address to y
LOAD  R2  R3       // Load y
INC   R3           // Increment
STORE R3  R2       // Store the new value to memory
```

For deeper nesting, follow multiple static links.

Method call

```
void p1() {
  int x, y, z;
  ...
  z = p2(x+1, y*2);
  ...

}
```

```
int p2(int a, int b) {
  ...
  ...
  ...
  return ...
  ...
}
```

1. Transfer the arguments and the static link.
   Store the return address in a register and jump to code of the called procedure.

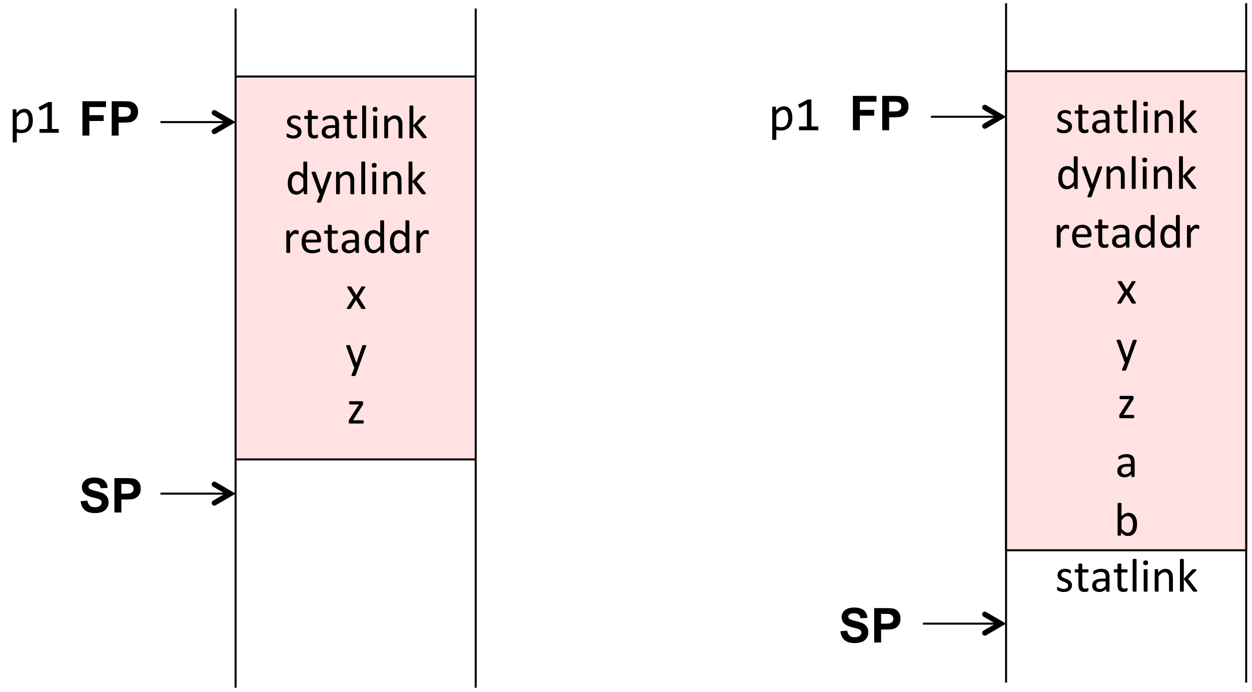
2. Allocate the new activation and move FP.

3. Run the code for p2.

4. Store the return value in a register.
   Deallocate the activation. Move FP back.
   Jump back to the return address.

5. Save the return value if needed.
   Continue executing in p1.

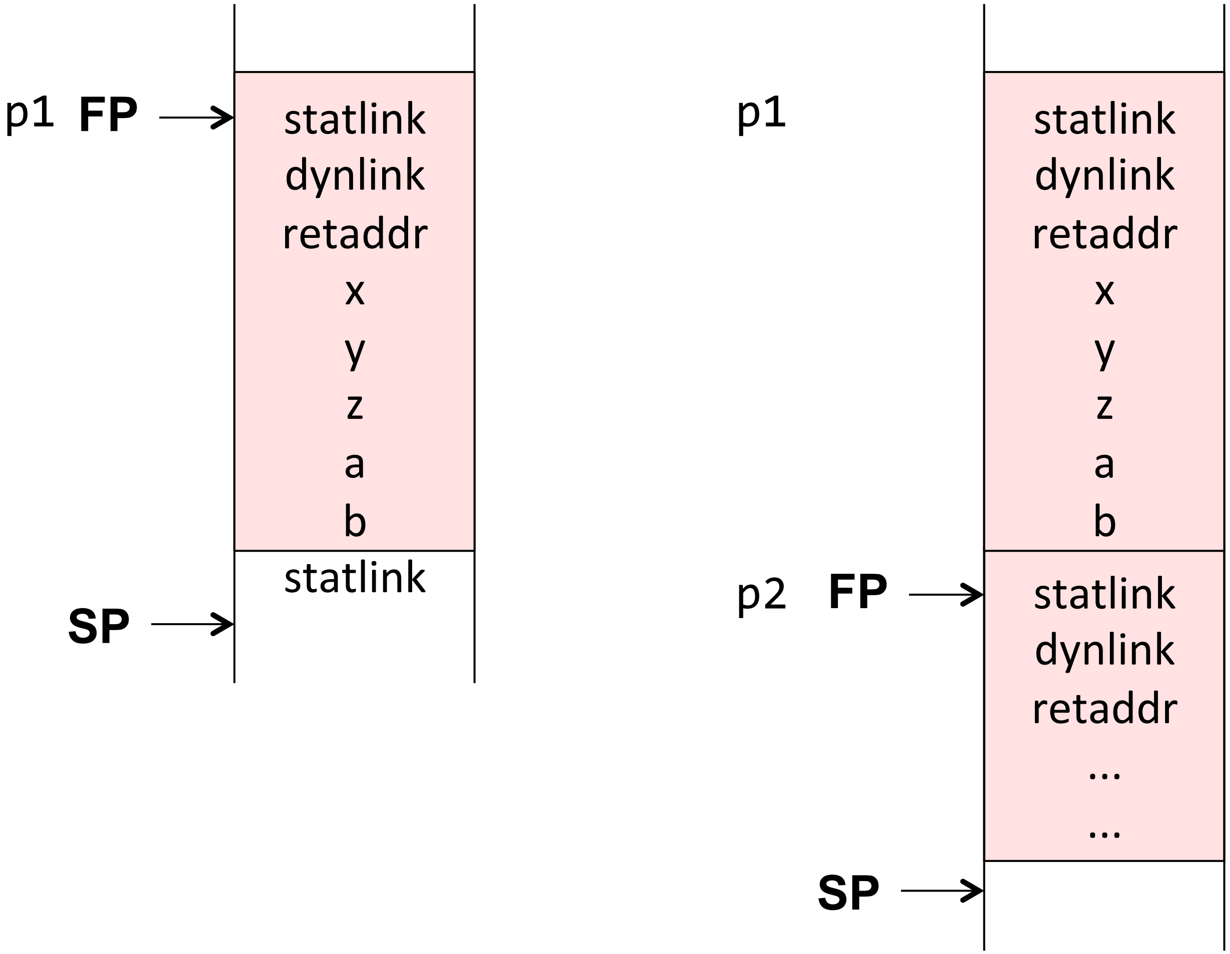
Step 1: Transfer arguments and call.



- Push arguments and static link.
  (The static link can be viewed as an implicit argument.)

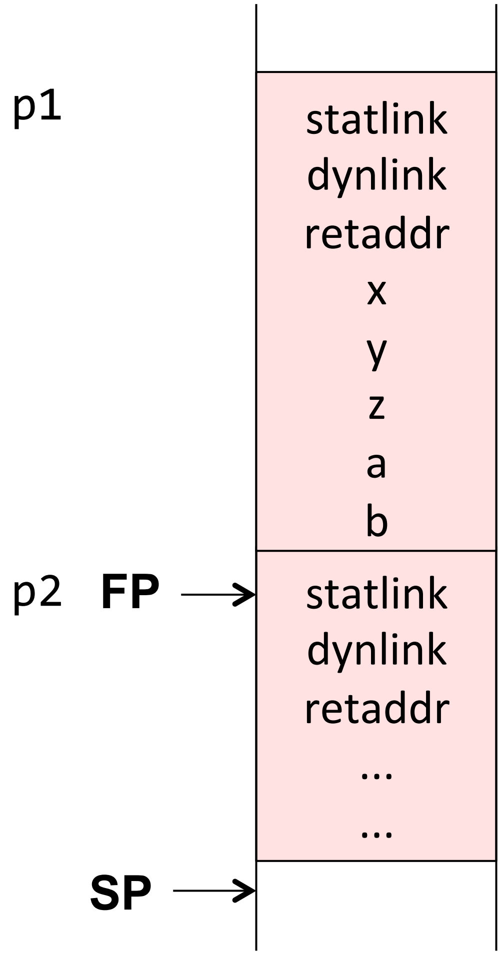
- Compute the return address (e.g., PC+4) and store it in a register.
- Jump to the called method code.
  (Usually use a CALL instruction for doing these two things.)

Step 2: Allocate the new activation



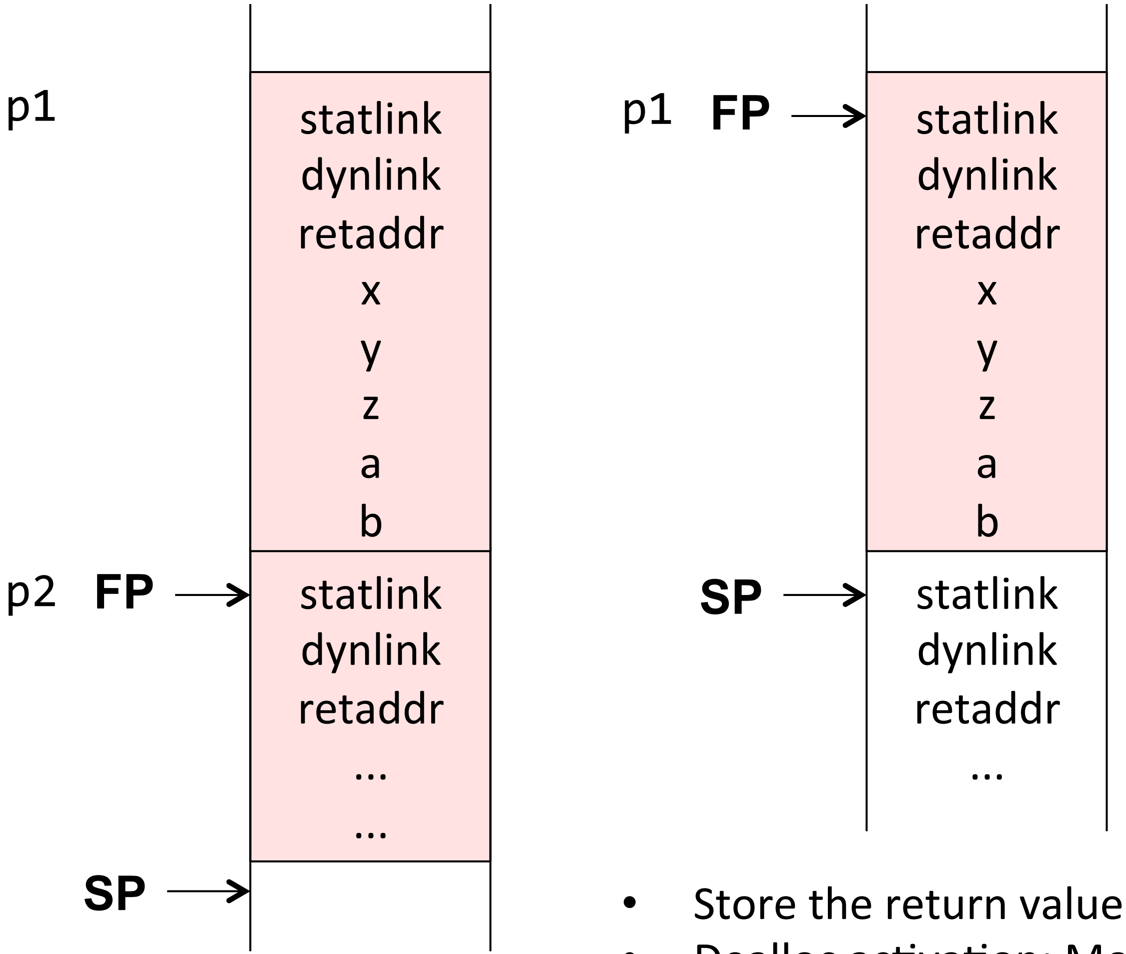
- push the dynamic link (current FP)
- set FP to the new frame
- push the return address (is in a register from the call instruction)
- push space for new variables and temps

Step 3: Run the code for p2



run the code for p2

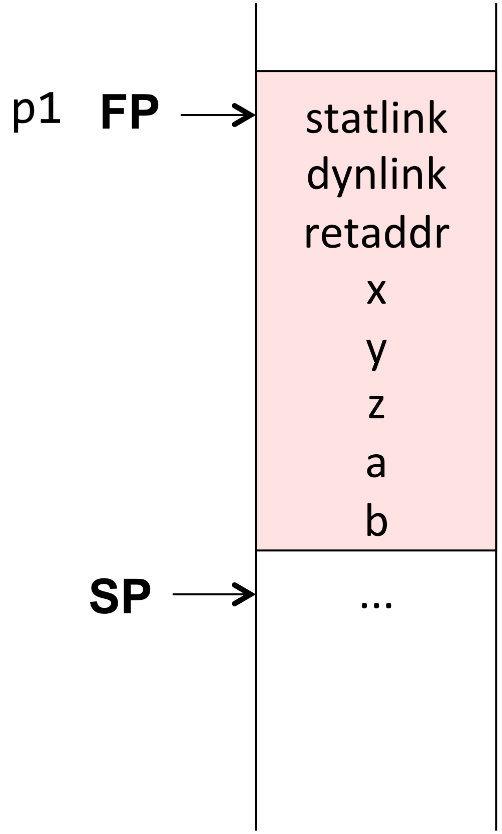
Step 4: Deallocate and return



- Store the return value in a register.
- Dealloc activation: Move SP back to FP.
- Move FP back by using the dynamic link.
- Jump back to the return address.

  (Usually do all this through a special RETURN instruction)

Step 5: Continue executing in p1



- Save the return value if needed.
- Continue executing in p1

What the compiler needs to compute

For variables and argument uses

- The offsets to use (relative to the Frame Pointer)
- The number of static levels to use (0 for locals)

For method calls

- The number of static levels to use (0 for local methods)

For method declarations

- The space needed for local declarations and temporaries

Registers typically used for optimization

Store data in registers instead of in the activation:

- The return value
- The *n* first arguments
- The static link
- The return address

If a new call is made, these registers must not be corrupted!

Calling conventions:
Conventions for how arguments are passed, e.g., in specific registers or in the activation record.
Conventions for which registers must be saved by caller or callee:

Caller-save register: The caller must save the register before calling.

Callee-save register: The called method must save these registers before using them, and restoring them before return.

Many different variants on activation records

Static link or not: Can treat it as an implicit argument when it is needed.
Dynamic link or not: Can let the compiler compute it for each method.
Stack pointer: Point to first empty word, or last used word?
Arguments: Treat them as part of the calling or called frame?
Argument order: Forwards or backwards order in the frame?
Direction: Let the stack grow towards larger or smaller addresses?
Allocate space for vars and temps: In one chunk, or push one var at a time.
...

Machine architectures often have instructions supporting a specific activation record design. E.g., dedicated FP and SP registers, and CALL, RETURN instructions that manipulate them.

Summary questions

- What is the difference between registers and memory?
- What typical segments of memory are used?
- What is an activation?
- Why are activations put on a stack?
- What are FP, SP, and PC?
- What is the static link? Is it always needed?
- What is the dynamic link?
- What is meant by the return address?
- How can local variables be accessed?
- How can non-local variables be accessed?
- How does the compiler compute offsets for variables?
- What happens at a method call?
- What information does the compiler need to compute in order to generate code for accessing variables? For a method call?
- What is meant by ”calling conventions”?