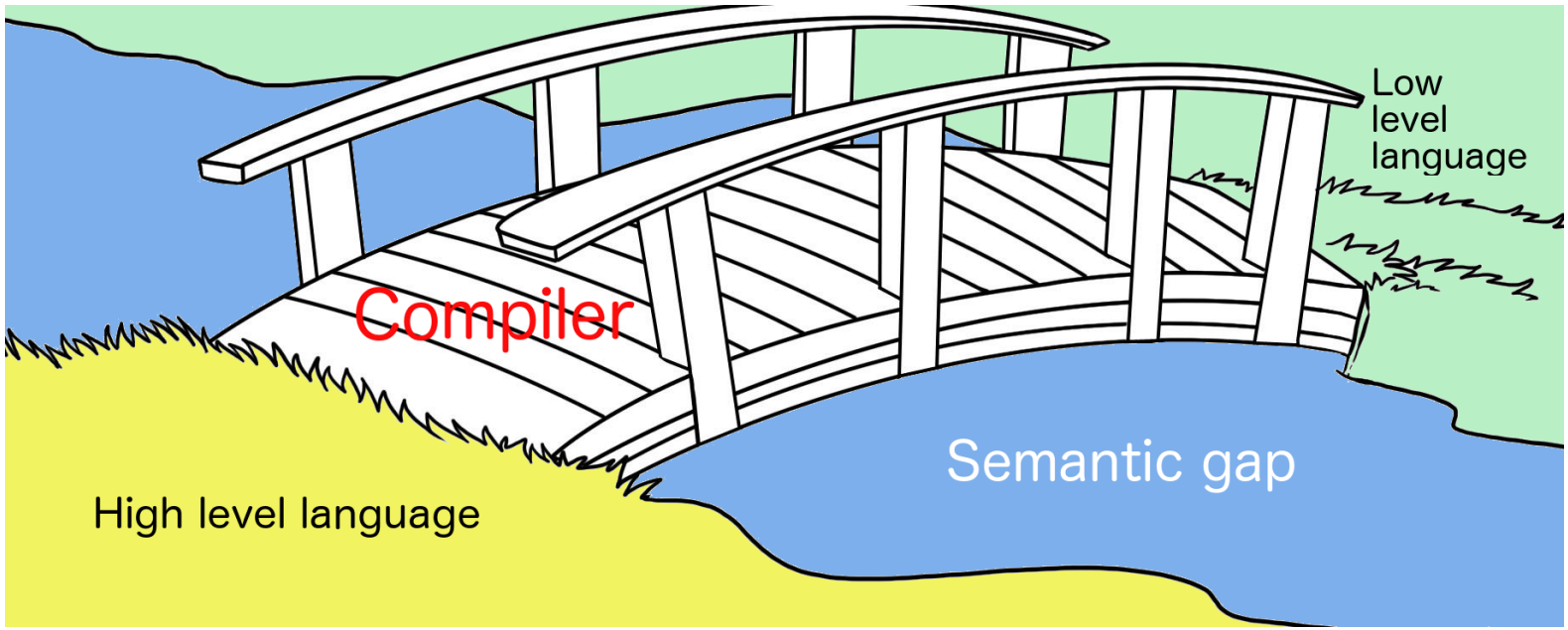
Compilers and computer architecture: Realistic code generation

Martin Berger [1]

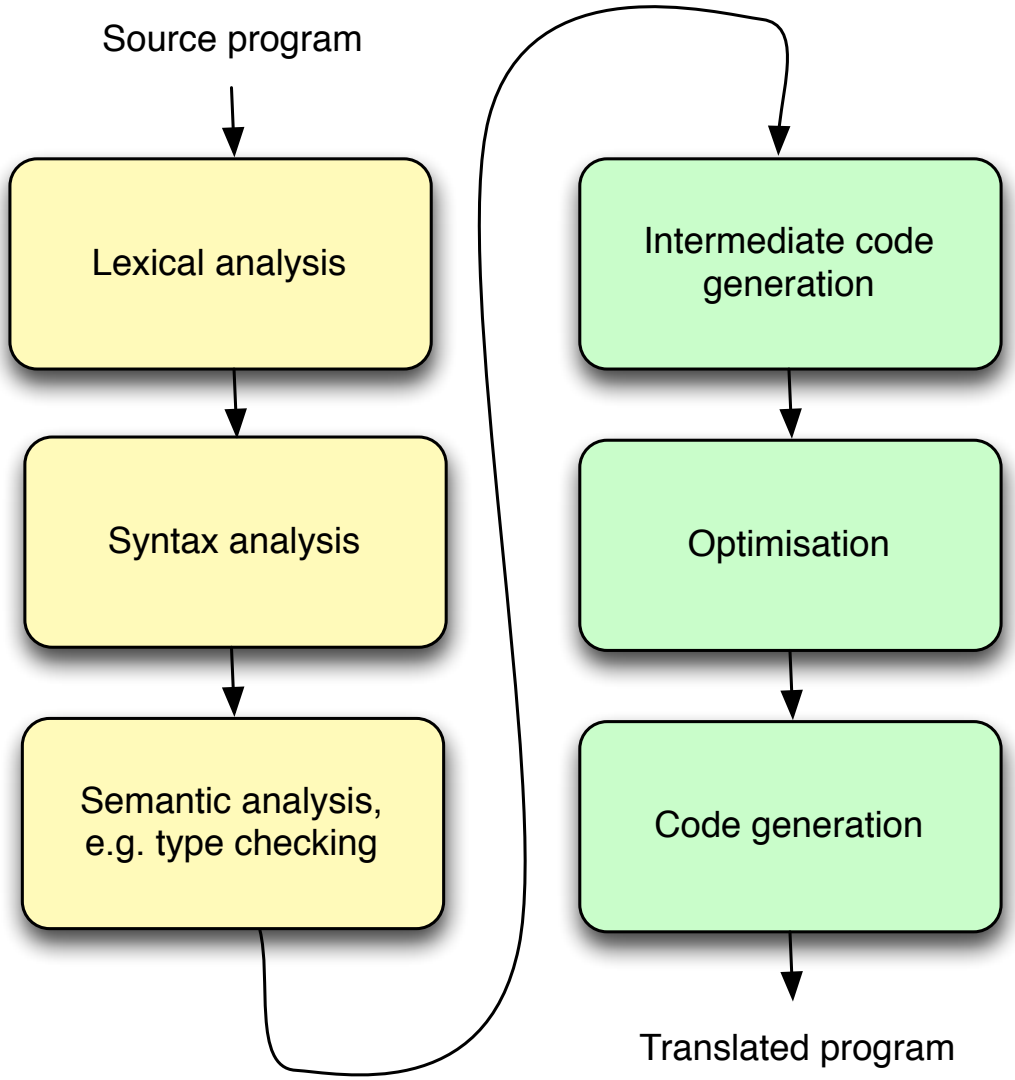
November 2019

[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall the structure of compilers



Introduction

We have 'finished' the compilers course, in the sense that we looked at all compiler phases: lexing, parsing, semantic analysis and code generation.

What you've learned so far is enough to make a basic compiler for a simple programming language.

Now we learn how to make a good compiler for a (more) sophisticated programming language.

To be able to do this, we must revisit code generation.

Introduction

We looked at code generation, but made various simplifications to highlight the key ideas.

- Source language was simplistic (no procedures, no objects etc).
- Target language was simplistic (e.g. stack machine, or register machine with unbounded registers).
- Translation produced slow executables (in the case of stack- and accumulator machines), also we looked at one simple way to improve register allocation.

Introduction

Now we look at more realistic code generation.

- Target is a 'real' architecture (e.g. RISC-V).
- Source language features e.g. procedures and classes/objects.
- Code generation is typically split into three parts:
  - Generation of intermediate code.
  - Optimisation.
  - Generation of machine code.

This leads to various complications.

Key issues in translation to a realistic processor: speed

The code generators we've discussed at the start of this course work fine, even on real processors. The problem is speed of executables.

Naive code generation schemes make many memory reference, that can be avoided with better generated code.

Why are memory references a problem? Because they are **slow** in comparison with register access (approx. two orders of magnitude slower).

But registers are **scarce** in real processors (core RISC-V has 32 integer registers). Putting a lot of effort into making best possible of registers pays off handsomely in executable speed.

Key issues in translation to a realistic processor: memory, energy

More generally a lot of effort goes into optimising the generated executable, not just for speed, but sometimes for memory, and, increasingly, for energy efficiency (important for battery powered devices, and for large-scale data centers).

For example on some architectures caching a computed value in memory and reusing it requires more energy than recomputing the value multiple times in registers.

Key issues in translation to a realistic processor: processor evolution

Processors evolve quickly. Extreme example: ARM processors in our phones.

In modern compilers up to 90% of compiler LOCs is optimisation. It would be annoying if we had to rewrite this every time a processor evolves a bit.

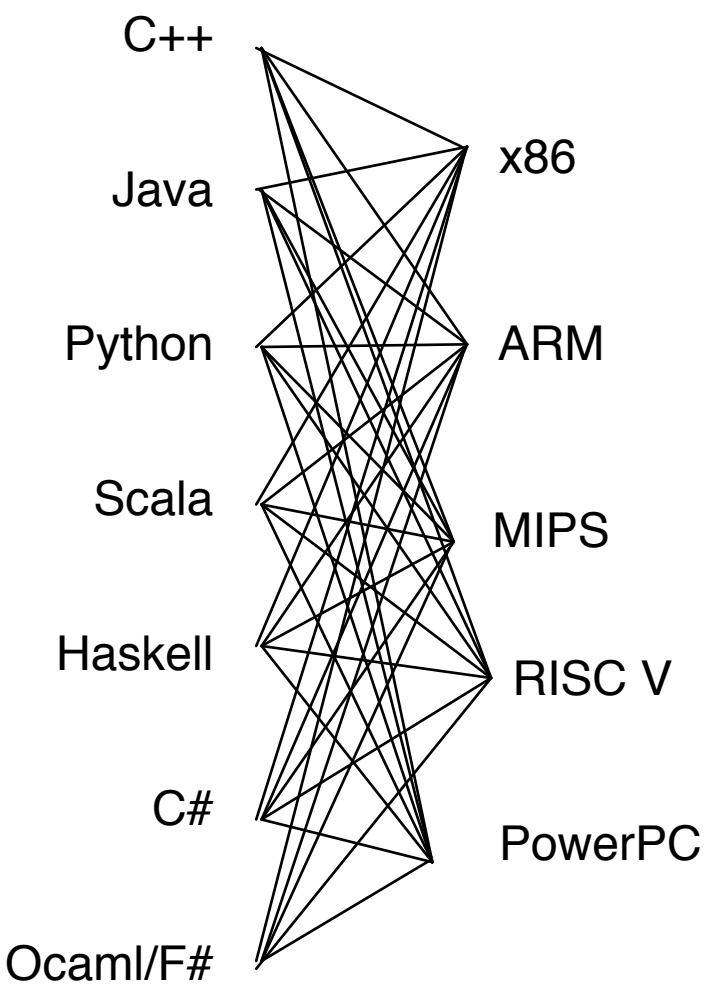
Fortunately, most machine languages are fairly similar.

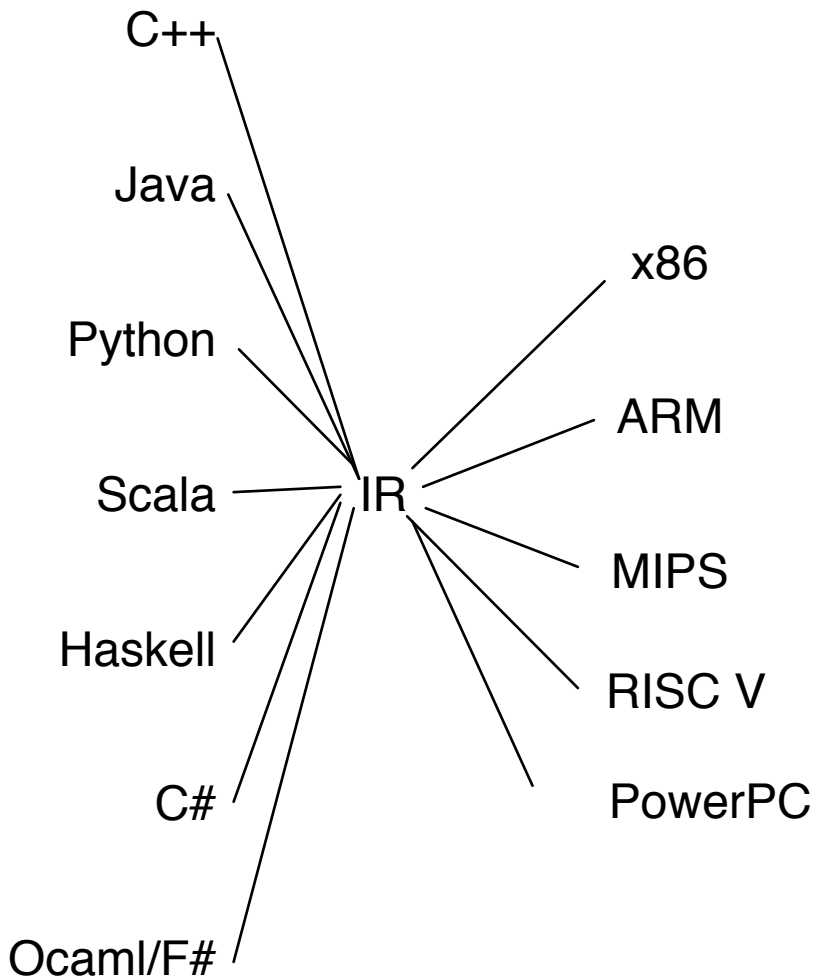
So we compile first to an intermediate representation (IR) which is rather close to generic machine code, but hides some detail.

- Most optimisation is done on this IR.
- Code for the target machine is then easy to generate from the IR.

A picture says more than 1000 words.

Key issues in translation to a realistic processor: processor evolution





Intermediate representation

A good intermediate representation should have the following properties.

- IR must be easy to generate for front-end.
- IR must be easy to convert to real machine code for all desired target machines.
- Each construct in the IR must have a clear and simple meaning, so that optimisation can easily be specified and implemented. Optimisations in real compilers are complicated and regularly introduce errors.

Real compilers often even use multiple IRs.

Intermediate representation: LLVM (not relevant for exams)

In 2019 the most well-known and widely used IR is probably LLVM (used to stand for “Low-Level Virtual Machine” although the project is now way bigger than virtual machines). It was developed by a student (Chris Lattner) for his final year dissertation. C. Lattner went to Apple where LLVM is now used in all Apple dev tools (OS X, iOS), did `clang`, Swift and was briefly head of Autopilot Software at Tesla. He’s at Google Brain now.

LLVM is free and open source, and has evolved into large tool suite for building compilers. I recommend that you take a look: `https://llvm.org`

If you want / need to build an optimising compiler, my recommendation is to use LLVM as backend.

Intermediate representation: WebAssembly (not relevant for exams)

Last few years saw the rise of compile-to-Javascript, e.g. TypeScript, Reason.ml and Scala.js.

Javascript is not a good language to compile to.

In the Summer 2017, all major browser vendors agreed to support WebAssembly as an platform independent IR.

WebAssembly's most interesting features:

- Typed
- Secure (e.g. no buffer overflow possible)

Will WebAssembly succeed? Time will tell ... but things are looking good.

Check out: `https://webassembly.org`

Intermediate representation

Often IRs (including LLVM) are quite simple and look much like the machine code for the register machine with unlimited registers.

Such IR is often called **three address code** because most instructions look something like this:

```
r1 := r2 op r3
```

Here `r1`, `r2`, `r3` are registers and `op` is an operation like addition, comparison ... It is clear that this can easily be mapped to real assembly instructions (i.e. a mini-compiler).

Intermediate representation: SSA (not relevant for exams)

Most modern compilers use a variant of three address code called **SSA** form, short for **static single assignment** form.

- Each variable is assigned exactly once (what about loops?)
- Each varianble must be defined (= assigned to) before use

This format makes compiler optimisations easier and faster.

Surprisingly close to functional programming.

Intermediate representation: PTX (not relevant for exams)

PTX (= Parallel Thread Execution) is an intermediate language used for compiling Nvidia's CUDA to Nvidia's GPUs.

It's a register machine architecture with unlimited registers.

Key issues in translation to a realistic processor: procedures, objects, methods

The key issues making code generation more complicated than the schemes we saw in the first few weeks are:

- Procedures, potentially recursively defined.
- Objects and subtyping.
- ...

Key issues in translation to a realistic processor: procedures

Procedures, potentially recursively defined, that can be invoked like this:

```
def f ( x : Int, y : Int ) : Int = {
   if ( x == 0 ) then 7 else f ( x-1, x+y ) }
...

f(2, f(3, f(4, 100)))
```

Problem: we don't know at compile time when, where and how often they will be invoked, so we must have dynamic (run-time) mechanisms that orchestrate the invocations.

Key issues in translation to a realistic processor: procedures, objects, methods

Objects and subtyping: We might have objects

```
class A { void f () {...} }
class B extends A { void f () {...} }
class C extends A { void f () {...} }
...
inp = read_user_input ()
A a = if inp == 0 then new B else new C
a.f()// how do we know which f to use?  B::f or C::f?
```

We must have a dynamic mechanism (i.e. working at **run-time**) that can make a dynamic dispatch at run time to the correct `f`.

End of overview

Recursive procedures are the tallest mountain to climb. The technology that allows us to compile recursive procedures also helps us to translate objects, exceptions etc.

Now that we understand the problems, let us begin looking at solutions.

First we look at the key ideas and data-structures (lifetimes, activation records, stack, heap, alignment) in a general way.

Then we will write a code generator using these ideas.

Management of run-time resources

Important terminology:

- **Static**, happens at compile-time.
- **Dynamic**, happens at run-time.

It's important to be clear what happens statically and what happens dynamically.

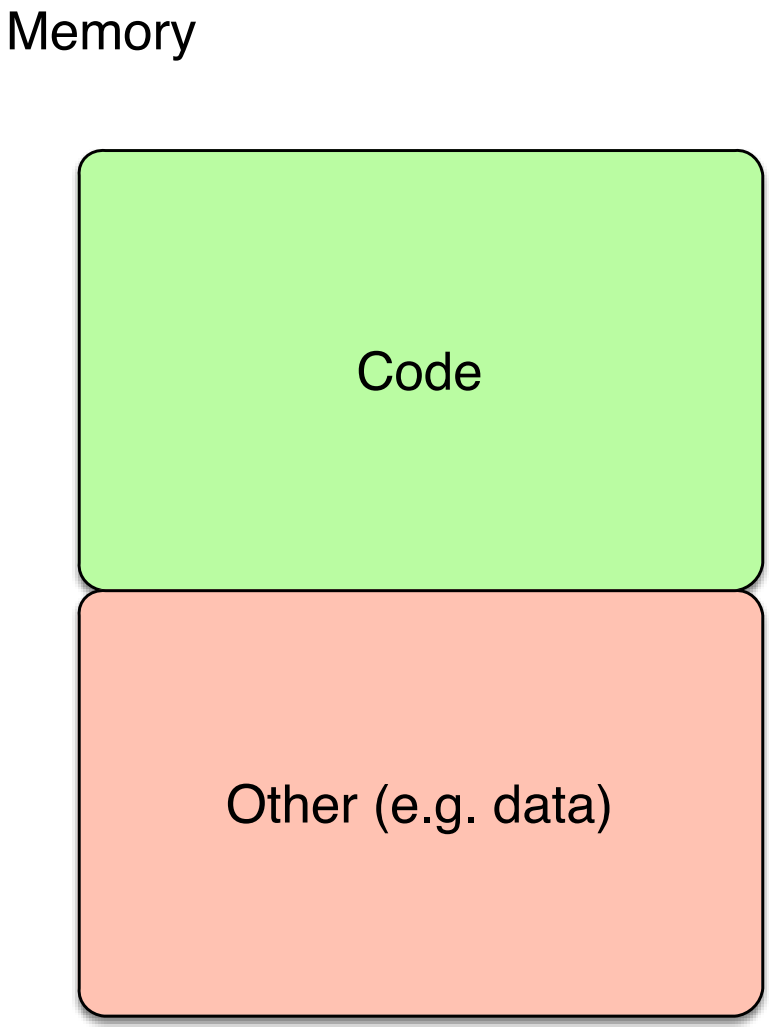
They interact: statically, the compiler generates code that dynamically organises computation (e.g. garbage collection, invocation of procedures/methods, allocation of heap memory).

Run-time organisation of memory

What happens when a program is invoked?

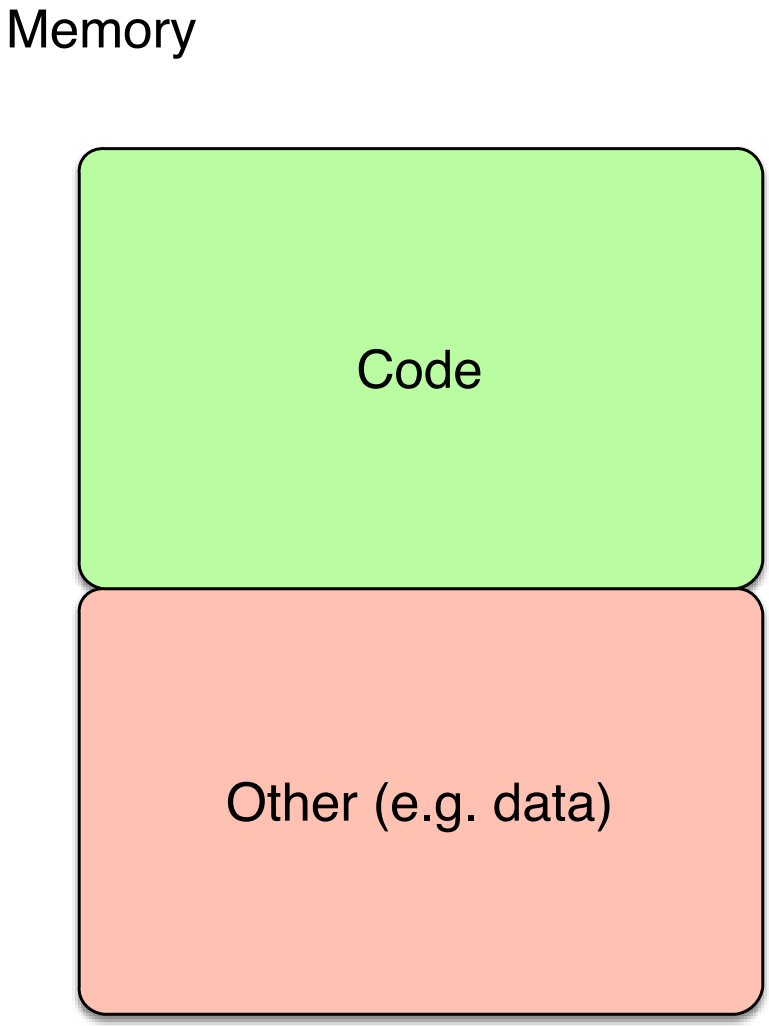
- We ask the OS to allocate space for the program.
- The OS loads the code of the program into the allocated space. Maybe maps symbolic names (remember those?) to real machine addresses. This is called (dynamic) linking.
- The OS jumps to the entry point of the program, i.e. the (translation of the) program's entry point (e.g.: in Java the `main` method).

Visualisation of memory



We often draw memory like this. Lines delimit different areas for different kinds of data. This a simplification, e.g. memory blocks might not be contiguous.

Runtime organisation



The compiler (at compile time) is responsible for

- Generating the executable code. This happens statically. Compiled code stays unchanged (i.e. is **read only**) after linking (with all modern compilers I know). Question: Why? Security, prevents attackers from modification (alas this is an insufficient defense).
- Orchestrating the dynamic use of the data area of memory.

Compilation of procedures

Now we are going to learn how to compile procedures (e.g. static methods in Java). The technology used here can also be used to compile (non-static) methods, but we need other technology in addition.

Note that we have two goals in code generation.

- Code should be **fast**.
- Code should be **correct** (= preserve meaning of source).

It's easy to achieve each goal separately.

All complication arises from trying to meet both goals together.

Compilation of procedures: key assumptions

From now on we base our discussion of code generation on two fundamental assumptions.

- Execution is sequential.
- When a procedure is called, control returns to the point immediately after the procedure call.

Compiling without either assumption is substantially harder.

Let's look at both in some more detail.

Compilation of procedures: key assumption sequential execution

We assume that there is no parallel or concurrent execution in our programming language. Only one thing happens at a time, and code is executed on step after the other.

Concurrency (e.g. Java threads, or multi-core CPUs) violate this assumption.

We will ignore concurrency.

Compilation of procedures: key assumption is simple control flow

We assume that when a procedure is called, and returns, the next command to be executed after the procedure returns is the command immediately after the procedure call.

```
x = x+1
y = f( x, 2 ) // assignment is executed after
              // f returns a value
z = y*z
```

We also assume that each procedure returns **at most once** for each invocation!

Languages with advanced control constructs `call/cc`, or `goto` or exceptions or concurrency violate this assumption.

We will ignore such advanced control constructs.

Two definitions: activations and lifetime of a procedure

Let `Proc` be a procedure.

An **activation** of `Proc` is simply an execution of a call to `Proc`, i.e. running `Proc`.

Activations have a lifetime. The **lifetime** of an activation of `Proc` is

- All steps to execute the activation of `Proc`, including ...
- ... all the steps in procedures that `Proc` calls while running (including recursive calls to itself).

Important observation of procedure lifetimes

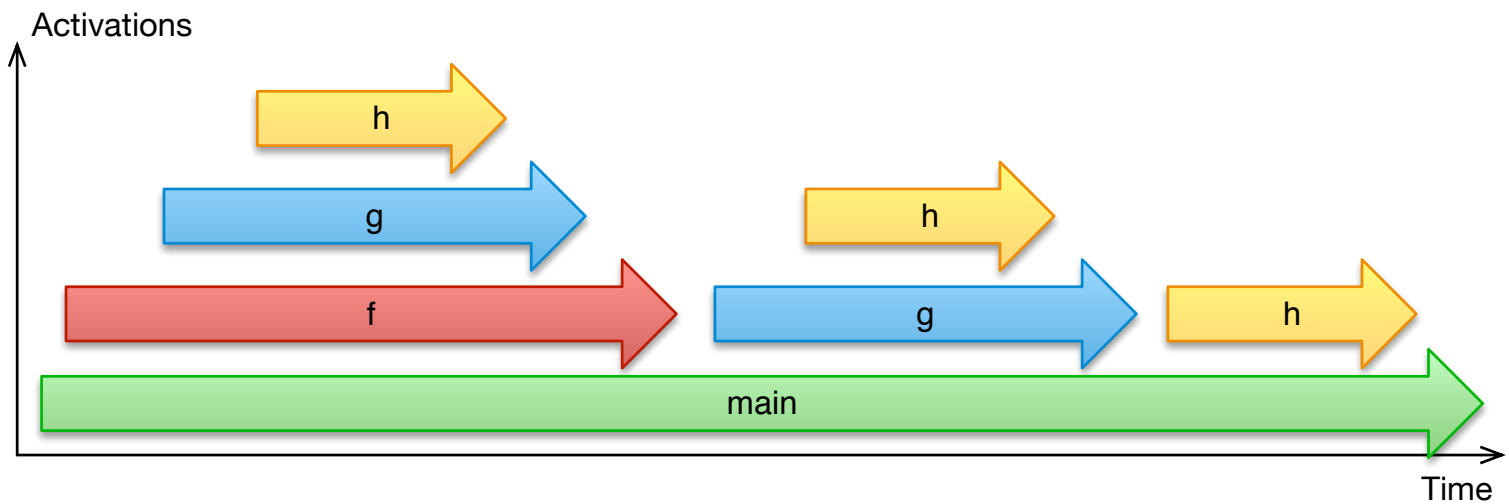
```
def f () = {
  println ( "entering f" )
  g()
  println ( "leaving f" ) }
def g () = {
  println ( "   entering g" )
  h ()
  println ( "   leaving g" ) }
def h () = {
  println ( "      entering h" )
  println ( "      leaving h" ) }
def main ( args : Array [ String ] ) {
    f()
    g()
    h()   } }
```

In class, `test.scala`, `test2.scala`.

Important observation of procedure lifetimes

Lifetimes of procedure activations are properly nested (well-bracketed):

When `f` calls `g`, the `g` returns before `f` returns, etc. Hence we can draw activations like below. This is a consequence of our two assumption above.



Important observation of procedure lifetimes

What about recursive procedures?

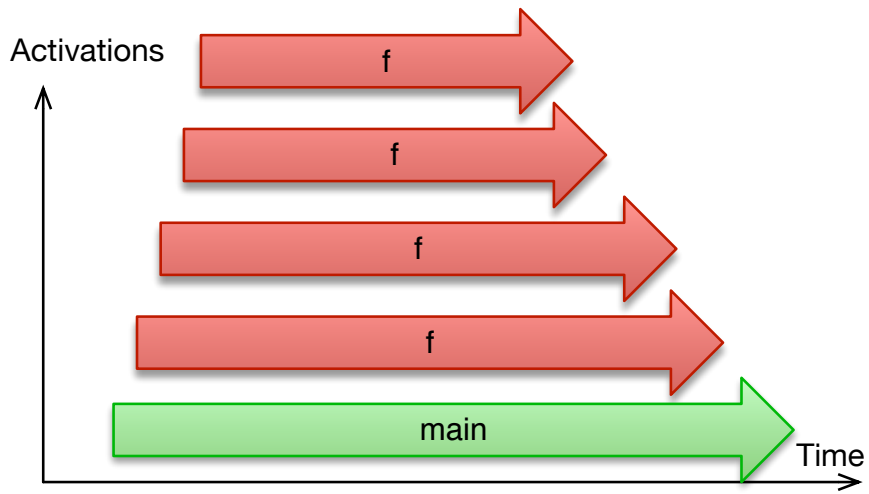
```
def factorial ( n : Int ) : Int = {
    println ( "entering f ( " + n + " )" )
    val result = if ( n <= 0 )
         1
    else
         n * factorial ( n-1 )
    println ( "leaving f ( " + n + " )" )
    result }

def main ( args : Array [ String ] ) {
    println ( "entering main" )
    factorial ( 4 )
    println ( "leaving main" ) }
```

In class, `factorial`, `factorial2`.

Important observation of procedure lifetimes

Again we see the nesting structure as a form of stacking.



Activations

We need to store some data to orchestrate execution of a procedure call. E.g. return address, procedure arguments, return value. Let's also call this information the **activation** ...

Where should we store this information, i.e. the activation, at **run time**?

Cannot use statically allocated memory address. Why? because more than one activation might be active at the same time (recursion) and we cannot predict how many are activations are active a the same time. Why? Recall Rice's theorem?

Activations

The activation tree of a program is dynamic, is run-time behaviour.

Cannot be predicted statically how many, and how they are nested (e.g. might depend on user input.) The activation tree may be different for different runs. **But they are always 'well-bracketed'**.

This suggests the following implementation of procedures: Use a **stack** to keep track of **currently active activations**, in order encountered:

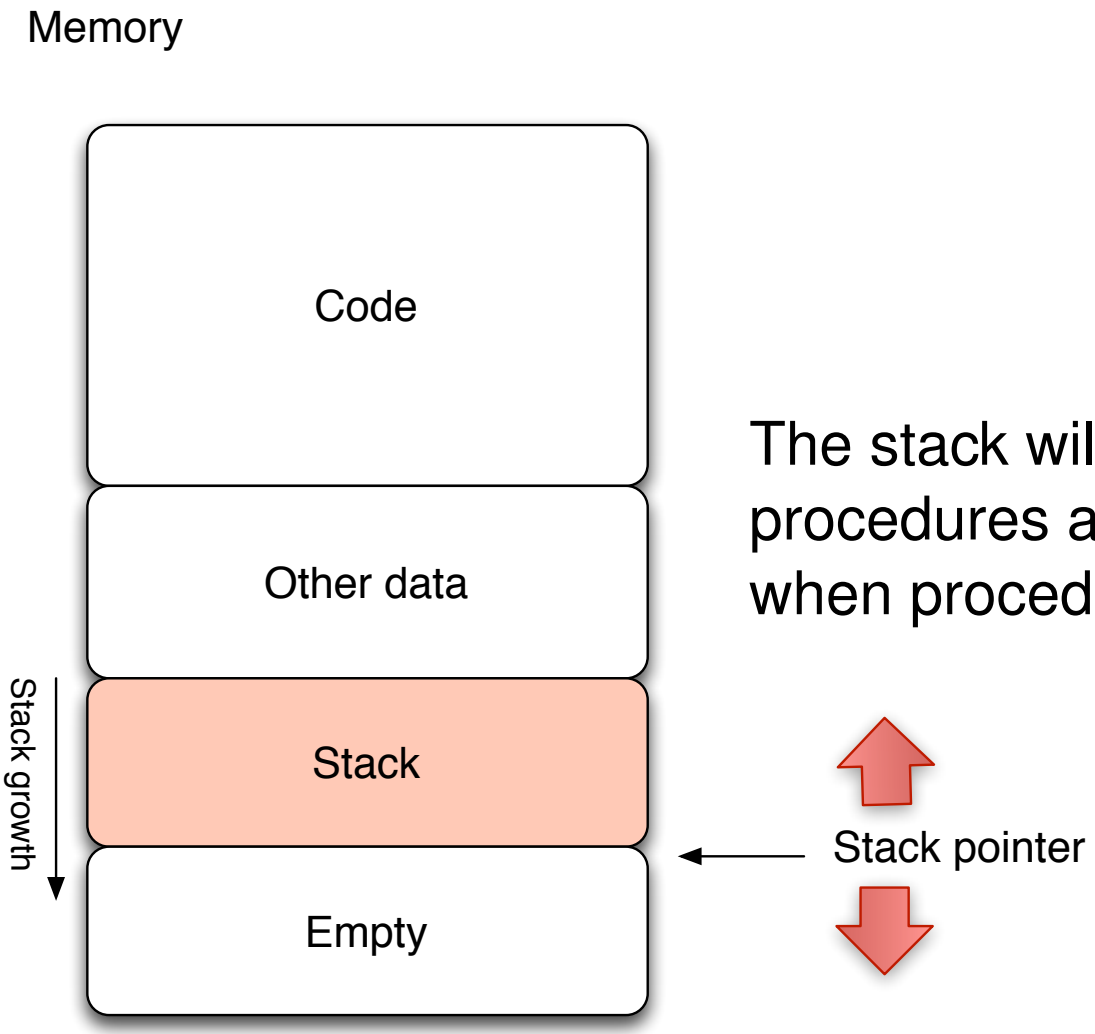
- When we call a procedure we create an activation on the stack, containing all relevant data about activation (eg. arguments, return address).
- When the procedure call terminates, we pop the activation off of the stack.

Example: stack stores activations

```
def f () = { g() }
def g () = { h () }
def h () = {}
def main ( args : Array [ String ] ) {
    f(); g(); f() }
```

Draw stack in class.

Memory organisation: stack



The stack will grow as new procedures are called, and shrink when procedure calls terminate.

Activation records, aka (stack)frames

Let's summarise.

An **activation**, also known as **activation record**, **frame** or **stack frame**, stores all the information needed to execute one procedure activation.

Activation records (and their being on the stack) are the key data structure to make procedures (and later methods) to work.

Activation records, aka (stack)frames

What information do we keep in an activation record?

Depends on the details of the programming language, compilation strategy, and target architecture. But roughly this:

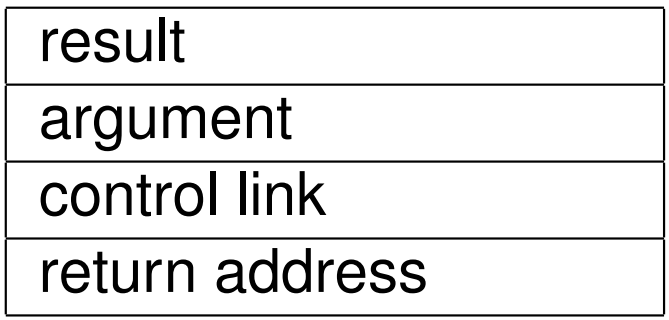
- Arguments for the procedure just called.
- Result of the activation, to be handed back to caller.
- Return address.

Activation records, aka (stack)frames

Let's look at an example.

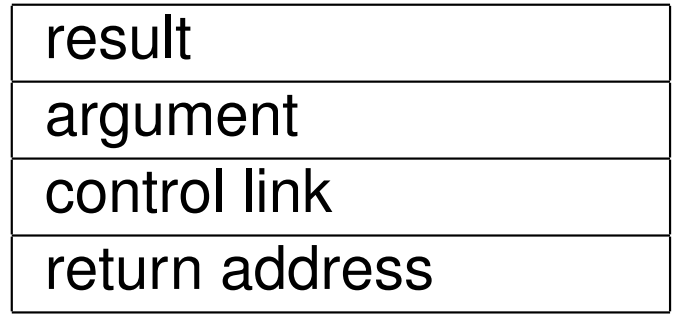
```
def g () : Int = { 1 }
def f ( n : Int ) : Int = {
  val result = if ( n <= 0 ) g() else n * f ( n-1 )
  result }
def main ( args : Array [ String ] ) {
  f ( 3 ) }
```

Here is a possible layout for activation records for `f` (for `main` and `g` they are different).



Activation records, aka (stack)frames

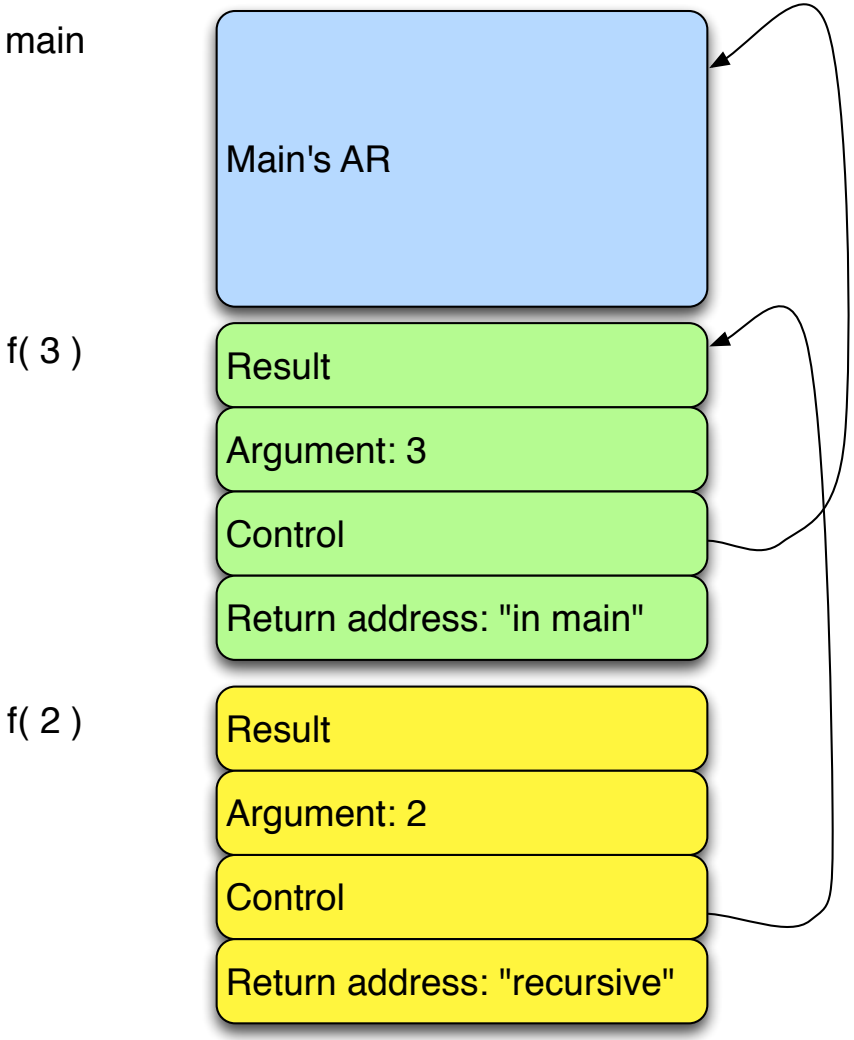
Here is a possible layout for activation records for `f` (for `main` and `g` they are different).



- The **result** holds the integer `f` returns.
- The **argument** holds the unique integer argument `f` gets. If a procedure has *n* arguments, then *n* slots are needed in the AR to holds them.
- The **control link** is a pointer to caller's activation record (explained later).
- The **return address** is where `f` jumps to when finished. Needed because `f` is called in multiple places.

Do you note something important about this? The size and shape (e.g. which field is at what offset) can be determined at **compile-time**. This has important consequences.

Example of activations record stacking



Main has no arguments and interesting return values, so AR is not so interesting.

“in main” and “recursive” denote the two places where `f` is called, so the call were execution resumes when `f` finishes.

Only one of many possible AR designs.

Stacks as arrays

Remember that we realised something very important earlier. The size and shape (e.g. which field is at what offset) can be determined at **compile-time**. This has important consequences:

The compiler can emit code that accesses any field in an AR, provided there is a pointer to the top of the AR.

But does the stack data type support this?

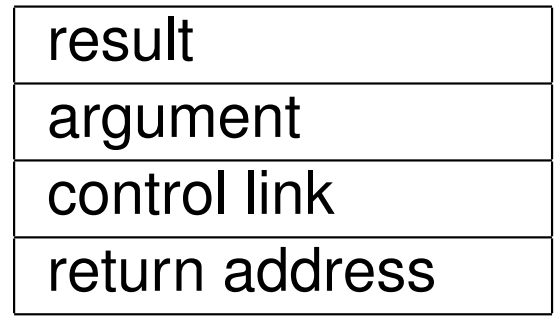
Stacks as arrays

Usually we think of the data-type stack as a black box that we can either push something onto, or pop off, no other operations provided.

Stacks in real CPUs support those, but are **also** big arrays.

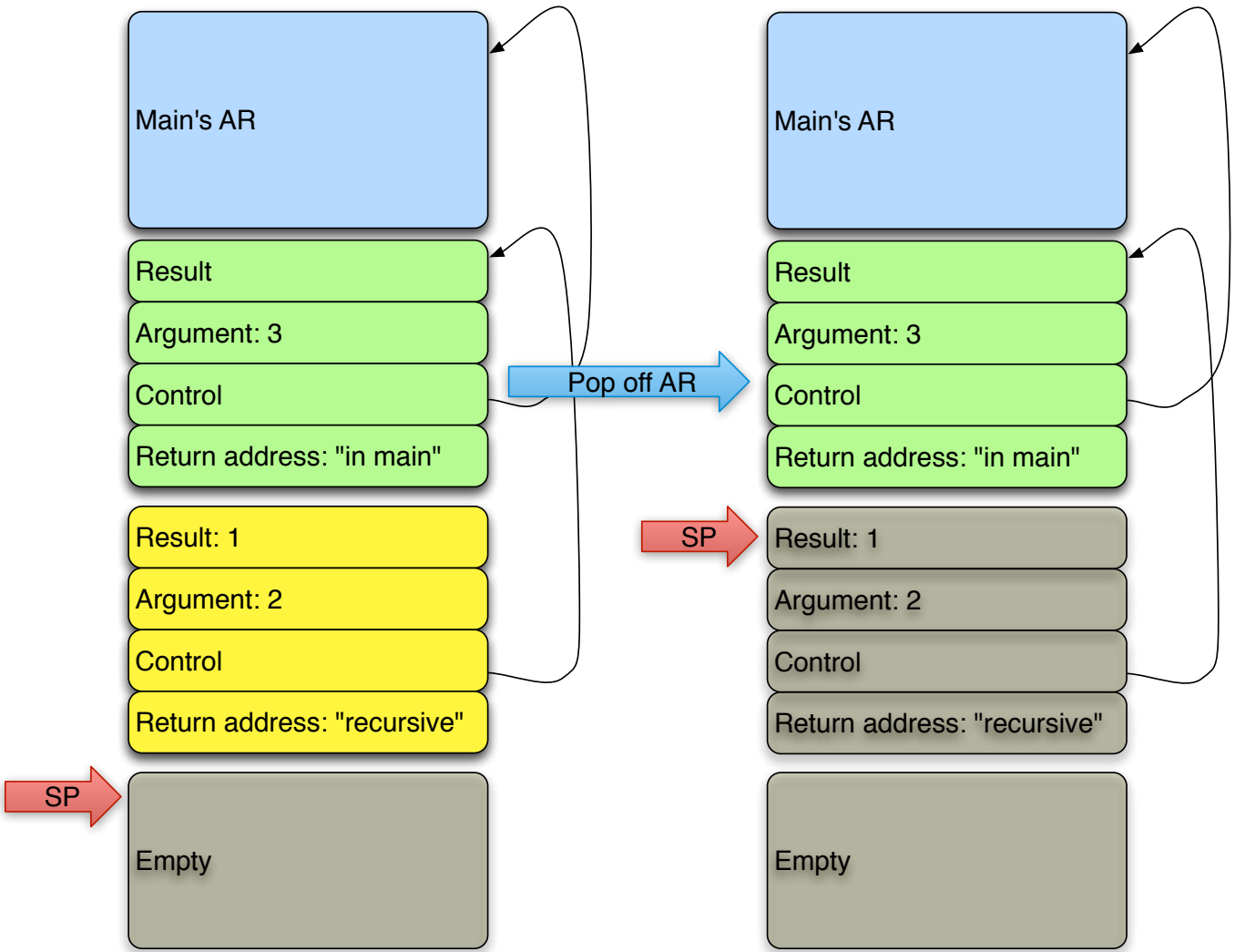
This means we access the fields in the activation record pointed to by the stack pointer relative to the stack pointer, e.g. `SP-4` or `SP-16`.

For example if the `SP` points to the top of the AR (i.e. to the return address) and each field is 32 bits, then the argument is at `SP-8` and the result sits at `SP-12`. If the `SP` points to the first free slot above the stack, then the argument is at `SP-12` and the result sits at `SP-16`.

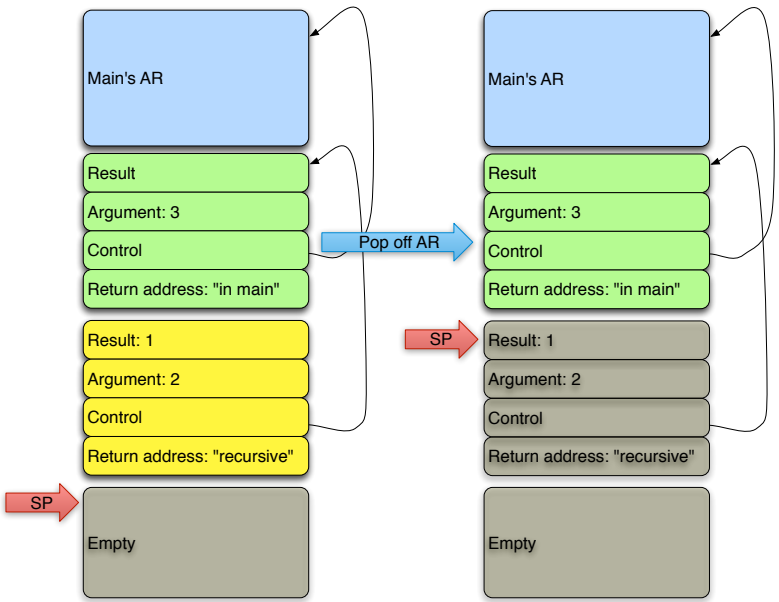


When AR is popped, SP points to result

Assume the `SP` points to the first free slot above the stack.



When AR is popped, SP points to result



Here we are using the fact that popping off something, only rearranges the SP, but data is physically still there (not overwritten). So the caller can easily access the result.

As we see later, local variables of procedures are also stored in ARs and are at fixed offset, so to access local variables we make use of the 'stack' also being an array.

Saving caller data

```
def f (...) {
    x = x*y
    g( x, y, z )
    x = z+y+z }
```

If a procedure `f` calls `g`, for example then the execution of `g` typically uses registers. But these registers are typically also used by the activation of `f` that is not yet finished.

It would be bad if the activation of `g` overwrote `f`'s registers.

What happens is that the call to `g` **saves** `f`'s registers before executing `g`, and when the execution of `g` is finished, these saved registers are **restored**. The saved registers are also stored in `g`'s AR.

The compiler must generate the code to save and restore caller registers. More about this later.

Note that the AR layout just sketched, and division of responsibility between the caller and callee are contingent (could have been done otherwise).

What conventions we use depends on many factors, such as:

- Compatibility with existing software.
- Availability of CPU support for some operations.
- Speed or memory usage.
- Code generation simplicity.

ARs in real compilers

Production compilers attempt to hold as much of the AR in registers as possible, especially the procedure arguments and result. In other words: in this case the AR is stored partly on the stack and partly in registers.

Reason: speed! Memory access is much slower than register access.

This can make procedure invocations complicated to compile.

Global and heap data

We've now begun to understand how a compiler handles procedure calls. We will come back to this soon.

But now we will look at global variables and heap data.

Global data

Consider the following Java program.

```
public class Test {
   static int x = 1;
   public static void main ( String [] args ) {
      System.out.println ( x ); } }
```

Where to store `x`?

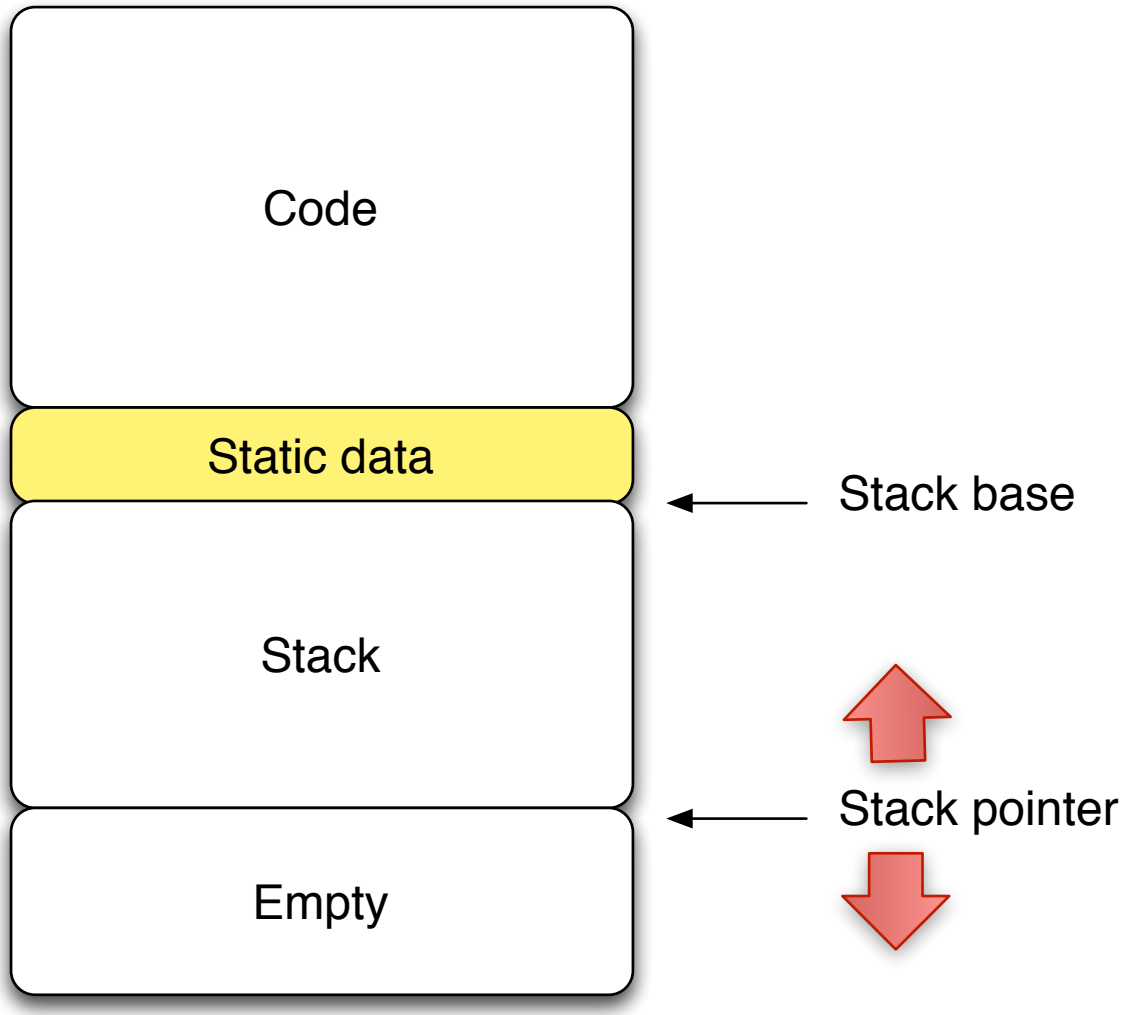
As `x` is **static** in the sense available from the start of the program to its termination, we cannot store `x` in an AR. That's because ARs live only for part of the program's lifetime.

We also say that `x` is **statically allocated**.

Depending on the programming language, other static data may exist.

Static data

To deal with static data, we augment the run-time layout of memory a bit by adding an area where static data lives throughout the lifetime of the program.



Heap data

Static variables are not the only thing that cannot be stored in ARs. Consider the following Java program.

```
public class A { ... }
public class B {
    A f () { return new A (); }
    void g () { A a = f(); ... } }
```

The object in red cannot be in ARs for invocations of `f` because it will outlive the invocation. The AR will hold a pointer to the object though.

Can we store the object with static data? Yes, but typically, the object will **not** be used for the program's entiere lifetime. The static data will not be garbage collected (see later), so memory there will not be reclaimed / reused. So storing the object there is wasteful.

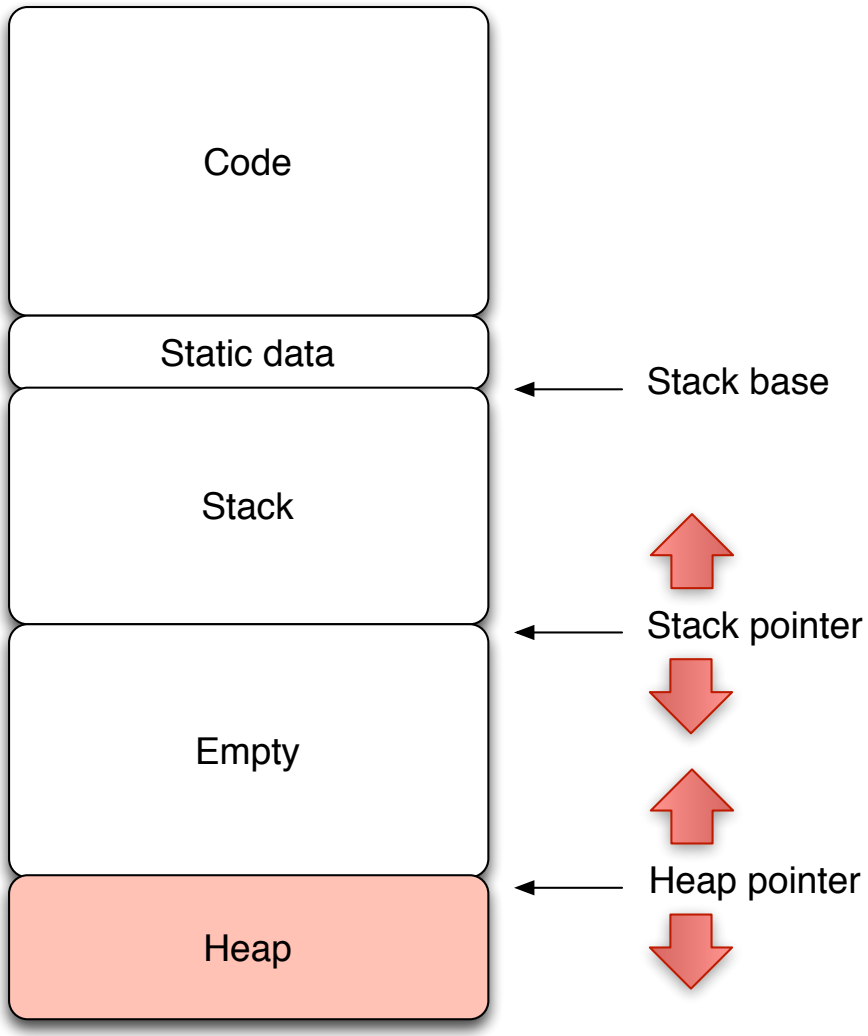
Heap data

So we need an additional memory region, where we store things that live **longer** than the activation where they are are created, but (typically) **shorter** than the program's lifetime. This region is called **heap**.

ARs are automatically popped off the stack when an activation of a procedure terminates, but how do we free memory in the heap?

Answer: garbage collection (e.g. in Java, Scala, Haskell, Python, Javascript), or manual memory management (e.g. C/C++). We will learn more about this later.

(Simplified) Memory layout



Code: fixed size.

Static data: fixed size, contains static variables.

Stack: dynamic, contains ARs.

Heap: dynamic, contains all other data.

(Simplified) Memory layout

Both heap and stack have dynamically changing sizes.

We must make sure they don't grow into each other.

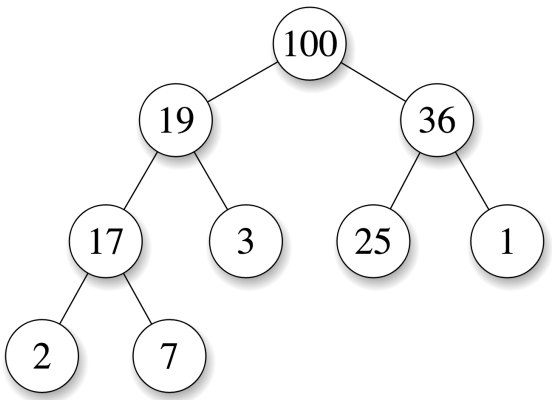
Solution: Stack and heap start at opposite ends of the memory and grow towards each other.

When stack and heap boundaries cross, we've run out of memory, and must terminate the program (or ask the OS for more memory). The compiler must generate code that checks for (and deals with) this. The OS can help with this (e.g. MMU).

Multiple meanings of “Heap”

A warning, the meaning of the term “heap” is different in other parts of CS.

In particular, in the study of algorithms/data-structures, a heap is a tree-based data structure, e.g. like so:



These heaps have nothing to do with the heaps in compilation.

Alignment

Here is another important (and annoying) issue to do with memory.

Most modern CPUs have a 32 or 64 bits data bus. That means the CPU reads 32 or 64 bits in one cycle.

But memory is addressed in bytes. This is for historical reasons.

Recall that a byte is 8 bits, and a word is 32 bits (= 4 bytes) in 32 bit machines, and 64 bits (= 8 bytes) in a 64 bit machine.

Alignment

Example: assume we have a 32 bit CPU that reads 32 bit words.

- The CPU reads a word at address 3000. Then it reads in one cycle the content of addresses 3000, 3001, 3002 and 3003.
- The CPU reads (or should read) a word at address 3001. Then it reads in one cycle the content of addresses 3001, 3002, 3003 and 3004.

Note that 3000 is divisible by 4, while 3001 is not divisible by 4.

Divisibility by 4 is important for many (most? all?) 32 bit CPUs because accessing 32 bits in memory starting at an address that is divisible by 4 is (much) faster than accessing memory at an address that is not divisible by 4. We say addresses divisible 4 are **(32 bit) aligned**.

Alignment

More generally, for a 2^n bit CPU (e.g. $n = 5$ means 32 bit, $n = 6$ means 64 bit), we say that addresses divisible by 2^{n-3} are **word-boundaries**. Memory access of 2^n bits starting at a word-boundary is **aligned**, access at all other addresses is **misaligned**.

- 32 bit CPUs, $n = 5$, so word boundaries, hence aligned access begins at an address that is a multiple of $4 = 2^{5-3}$, e.g. 0, 4, 8, 12, 16, 20, ...
- 64 bit CPUs, $n = 6$, word boundaries are at addresses that are multiples of $8 = 2^{6-3}$, e.g. 0, 8, 16, 24, 32, 40, ...
- 128 bit CPUs, $n = 7$, word boundaries are at addresses that are multiples of $16 = 2^{7-3}$, e.g. 0, 16, 32, 48, 64, ...

Important: misaligned access is much slower (approx. 10 times) than aligned access for many (most? all?) CPUs. In some CPUs misaligned access is an error.

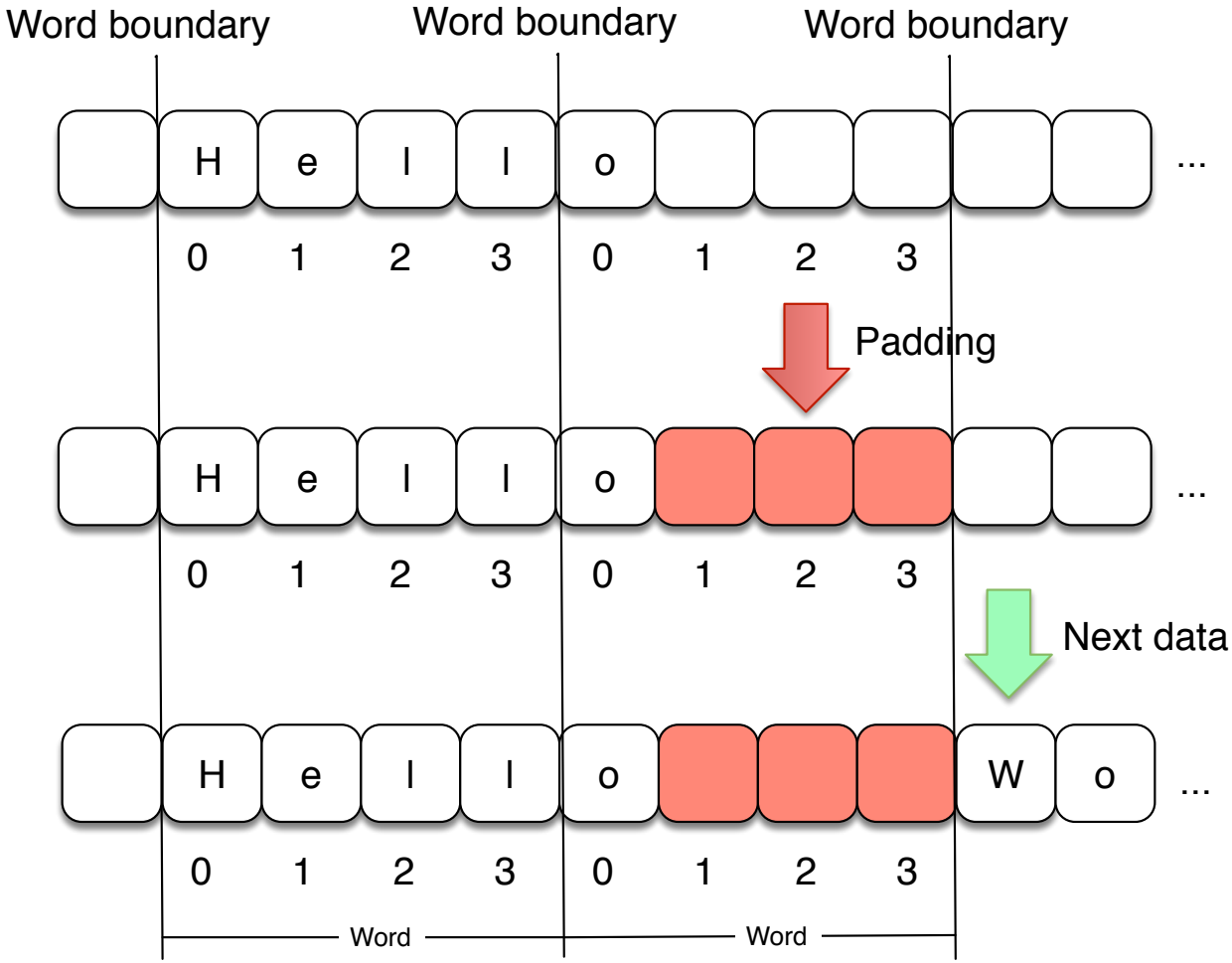
Alignment

Because of the huge speed penalty for misaligned memory access, compilers must do their best to ensure that all memory access is aligned.

Compilers achieve this by always locating data at word boundaries, using **padding** where necessary.

Alignment

Here is an example of padding data for a 32 bit CPU. We want to store the strings “Hello” and “World”.



Alignment

Most assembler languages have build in support that aligns (some) data automatically.

For example RISC-V assembly language has the `.align` command. Putting `.align n` in a line means that the succeedings lines are aligned on a 2^n byte boundary.

```
.align 2
.asciiz "Hello world!"
```

Here, `.align 2` aligns the next value (the string "Hello world!") on a word boundary on a 32 bit machine.