Code shape, code generation

- generally a huge range of possibilities for representing the original program as code in the target language
- tradeoffs: runtime speed, memory use, number of registers used, amount of energy used during execution...
- code shape: term used to cover the broad range of decisions that impact the target-language code eventually produced
- can include decisions in the intermediate/internal representation, since those can heavily impact ability to make effective decisions at the code generation phase

Storage: revisited

- overall storage model chosen: memory driven model (keep things in memory as much as possible) vs register driven (keep in registers as much as possible)
- decisions on where/when items reside in memory (static space, stack space, heap space)
- compiler needs to adhere to processor/OS handling of memory: usually granted a logical memory space per program, which OS maps to physical space (preventing one program from accessing memory space of another)

Typical layout of logical memory

- area for executable code storage
- area for static storage (possibly divided into areas for global variables, global constants, static local variables...)
- area for run-time stack
- area for heap
- heap and stack often located at opposite “ends” of free memory space, growing toward one another (allowing flexibility for which/when grows faster/larger)

Compiler freedoms

- source/target languages generally have strict rules regarding scope and storage of variables/data
- compiler free to create additional data items (in static space, on stack, in heap) to aid in program admin and for optimization (e.g. storing a computed value that will be used again later)
- compiler also free to arrange data in ways not visible to the programmer, e.g. try to improve performance by optimizing cache hit ratio, or try to minimize wasted space

Example: compiler and cache hits

- OS generally has memory divided into pages, keeps a set of frequently-accessed pages in high speed cache (or layers of caching)
- compiler might try to ensure that data values that are frequently used “close” to one another during execution wind up on the same page, thus improving cache hit ratio
- gets trickier when activation record large enough that it likely spans multiple pages

Example: minimizing waste space

- compiler must obey OS/hardware memory alignment rules (e.g. 1-byte items like chars can be at any address, 2-byte items like shorts must start on an even address, 4-byte items like ints must start on an address divisible by 4, etc)
- Programmer might provide lists of variables, parameters, or struct/class fields in orders that don't align with this naturally, e.g. char w; int x; char y; int z;
- Even if x happens to land on address divisible by 4, that means z's “natural” starting address would be odd

Space and alignment cont.

- compiler can insert padding (wasted space) in between variables to get the alignment right yet preserve original order of variables
- compiler can rearrange variables in memory, going from most-restricted to least-restricted w.r.t. alignment (e.g. all the longs/doubles/pointers first, then all the ints/floats, then all the shorts, then all the bools/chars) ... saves space, but internally the order may not be what programmer expects (only relevant if programmer is accessing through offsets)

Intermediate representation

- Will assume a register-driven model, each value gets its own virtual register, copied to/from the register at entry/exit points for the relevant scope
- relies on a good register allocation algorithm later to map these virtual registers to real physical ones, but makes for simpler IR and good possibilities for optimization

Names: ambiguous, unambiguous

- Code refers to memory locations by names frequently, e.g. x, y, arr[i,j], (*ptr), etc
- some of these names are unambiguous: can only refer to one possible storage location, and hence the storage they're referring to can be kept in a register
- e.g. variable x can be kept in a register (again, with suitable copies to/from x on scope exits/entries)

Ambiguous references

- Some named references are ambiguous, not clear at compile time which storage location they'll refer to
- arr[m,n] and arr[i,j], or (*x) and (*y) ... can we be sure they're not referring to the same storage location?
- `*x = i + j`
- `*y = 1`
- `z = *x + *y`

if x==y then z will be 2,
otherwise z will be i + j + 1

- If we were keeping (*x) in a register, and (*y) in a register, we can't be sure they're meant to be two different locations

Dealing with ambiguity

- compiler could insert code that will resolve ambiguity at runtime (e.g. Code to test if x == y first, then update both registers on the assignment to y)
- compiler could simply put values back in memory where ambiguous references involved
- either choice involves generation of some additional runtime operations, each will be superior in certain circumstances