CS 326 Final Review

Alex Zorzella

May 2025

Contents

1 Chapters 7 & 8: Data Types

1.1 Type Equivalence

1. Structural equivalence: same internal structure
2. Name equivalence: Alias example: `type cat = record {...}; type feline = cat;`, see equivalence below
   (a) Strict: aliases are distinct
   (b) Loose: aliases are equivalent

Which of the following types are equivalent?

```
type student = record
    name, address : string
    age : integer

type school = record
    name, address : string
    age : integer

type college = school
```

(a) Structural: student, school, and college are all equivalent

(b) Strict name: student, school, and college are all distinct

(c) Loose name: only school and college are equivalent

1.2 Type Conversion, Compatibility, and Casts

1. Conversion/casting is explicit, written in the code. The explicit keyword denotes the intentionality.
2. Coercion is implicit and done by the compiler when it sees fit. Compatible variables are coerced at compile time.
3. Non-converting cast

   Instead of casting explicitly, pointer manipulation can be used to reinterpret the data as another type: `*((float *) &n);`.
4. Compatibility

   In most languages, variables just have to be compatible to interact with each other, i.e.

   ```
   var a, b : real;
   var c : integer;
   ...
   a := b + c;
   ```

   is valid as `real` and `integer` are compatible.

   Stricter languages like Ada have various rules around type compatibility but as a result are more stable: type S is compatible with type T if and only if

   (a) S and T are equivalent
   (b) One is a subtype of the other
   (c) Both are subtypes of the same type
   (d) Both are arrays, with same numbers and types of elements in each dimension

   While C allows coercion of various types into other types but these operations are susceptible to truncation (rounding; data loss).

1.3 Data Types

1.3.1 Records

Records are simple data structures. All members of records are public, and they may not contain functions. They're very useful for when trying to store a small amount of data in conjunction without the overhead of a class. Their data may be stored in various ways. In the simplest case, the data is stored in intervals according to the largest datatype in the record. While this is efficient to access as the memory is sequential, it often creates holes and wastes memory.

A variant of the vanilla record is the packed record (available in Pascal, and not to be mistaken with Variant Records). Packed records store their data in the minimum amount of space possible, but in turn sacrifices computational power as accessing different members take multiple instructions.

Records can have various storage rules. Remember: if a record has a rule that specifies divisibility at a certain location, that record must follow that rule even when chained with itself in memory. If a record won't chain properly, then pad the end with empty bits until it does.

While a record can be "copied", i.e. `new_cat := calico`, copies of all elements in the record are shallow. Datatypes like `int`s, `float`s, and `string`s may be copied over just fine, but pointers won't deep copy and therefore reference the same region of memory as the original record. This can lead to unintentional mutation.

Record comparison may lead to false negatives as padding may contain garbage values, making their bit sequences differ. While records can fill padding memory with zeroes at initialization, doing so is computationally inefficient.

In Pascal, the `with` keyword can help simplify initialization or variable population. For example,

```
ruby.chemical_composition.elements[1].name := 'Al';
ruby.chemical_composition.elements[1].atomic_number := 13;
ruby.chemical_composition.elements[1].atomic_weight := 26.98154;
ruby.chemical_composition.elements[1].metallic := true;
```

can be simplified to

```
with ruby.chemical_composition.elements[1] do
begin
   name := 'Al';
   atomic_number := 13;
   atomic_weight := 26.98154;
   metallic := true
end;
```

Since C doesn't have a `with` operator, the same effect can be achieved with a pointer

```
p = & ruby.chemical_composition.elements[1];
    p->name = 'Al';
    p->atomic_number = 13;
    p->atomic_weight = 26.98154;
    p->metallic = 1;
```

Variant Records are records that use the same space to store either one of two different fields. These fields are mutually exclusive and never overlap, saving on space. For example, if a variant record either stores information for a cat or a dog, the two fields `lives:int` and `bones:real` can be stored in the same chunk of memory. Cats will never have bones and dogs will never have more than one life, so space is saved and cats and dogs don't have to be stored in separate records. Due to the lack of a discriminant for that field, there's no way of knowing what's stored in there for sure.

1.3.2 Arrays

Arrays are par for the course. They're data of the same type stored sequentially in memory. The shape of an array dictates the number of dimensions and bounds for the array. Since the amount of memory required to allocate may not be known at compile time, there are a couple of cases:

1. Global lifetime, static shape: global variables in C and Pascal are allocated statically at compile time
2. Local lifetime, static shape: local variables in C and Pascal are allocated dynamically at runtime
3. Local lifetime, shape bound at elaboration time: local variables in Ada are allocated on the stack at runtime
4. Arbitrary lifetime, shape bound at elaboration time: Java arrays are allocated with new and once allocated, maintain their shape until deallocation. This occurs dynamically on the heap at runtime
5. Arbitrary lifetime, dynamic shape: Perl arrays can have their shape changed during its lifetime at runtime. If the size of the array is increased, then a new block is allocated, the data from the old block is copied over, and the old block is deallocated. This occurs dynamically on the heap

One dimensional arrays are allocated contiguously in memory. Multi-dimensional arrays, however, can be allocated in two different ways: contiguously or row pointers. Contiguous allocation works the same in higher dimensions, and modulo operator can be used to calculate the staring point of each row. Large multi-dimensional arrays can take up large portions of memory which can lead to fragmentation. They also can be stored as row pointers in which the head of each row is stored in the first column of the array, via which each column of the rows can be accessed.

$$\text{Contiguous row access equation: } (i * S1) + (j * S2) + \text{address of } A - [(L1 * S1) + (L2 * S2)]$$

Fact: Fortran is the only language which stores contiguous multidimensional arrays in column-major order (top to bottom, left to right). Every other language stores them in row-major order (left to right, top to bottom). Traversing consecutive elements in consecutively stored arrays maximizes cache hits as the elements will be stored in the same general region of memory.

Multi-dimensional arrays avoid fragmentation, but consume more space as the pointers have to be stored in the first column alongside the data actually stored inside of them. It does lead to faster access times (machines in the 70s were slow at computing multiplications) and each row can store arrays with different lengths.

1.3.3 Pointers

Pointers, again, are par for the course.

Note: pointers are NOT the same thing as addresses.

1. Pointers are high-level concept (abstraction)
2. Addresses are a low-level concept (part of the implementation of memory)
3. Examples: segmented memory (segment ID and offset within segment), catch dangling references (address and access key)

Dangling pointers vs garbage: garbage is created whenever a pointer is deallocated. Depending on the language, the garbage left over might be automatically dealt with by a garbage collector or it may need to be explicitly dealt with. Dangling references exist when two or more pointers point to the same region of memory and one of those pointers deallocates the memory. Every other pointer will still have a notion that something exists wherever it points to and will throw an error if attempted to be dereferenced.

There are a couple of solutions to dangling pointers:

1. Tombstones Tombstones are the middle man that keep track of whether a region in memory hasn't been deallocated. Pointers that would have pointed to that region of memory instead point to the tombstone which will let the pointers know the status of the data. Unfortunately, it means that there's more overhead during pointer allocation and requires a validity check every time the pointer is dereferenced.

2. Locks and Keys In a lock and key strategy, pointers have a key value that can be compared to the value lock associated with a block of memory. If the memory becomes deallocated or overridden, then lock associated with it has its value reset or overridden which safely denotes that that region of memory is no longer valid for the other pointers associated with it.

1.3.4 Garbage Collection

1. Explicit deallocation of heap objects:

   The call is guaranteed to be faster as there's no runtime overhead, but the burden is on the programmer which can lead to memory leaks.

2. Automatic deallocation/garbage collection:

   The calls are automatic and trade the burden on the programmer for more complex implementation which means runtime overhead.

1.3.5 Pointer-Array Duality

The name of an array points to its first location in memory. Observe:

```
int *a;
int b[5] = { 1, 3, 5,  7,  9 };
// mem add: 100 104 108 112 116
a = b;

a        => 100
a[0]     => 1
&a[0]       => 100
a+2       =>   108
*(a+2)   => 5
(a+2)[1] => 7
```

There are a couple of ways to manage garbage collection:

1. Reference Counts

   Store the number of references to a region in memory, then deallocate it when there are zero. Regions in memory might have pointers to other regions, so whenever an object would become deallocated, `delete` is called on all subpointers of a pointer. This strategy is susceptible to circular references, however, as to regions in memory can reference each other. This ouroboros of references can be solved with the mark and sweep method.

2. Mark and Sweep

   The mark and sweep is a separate method from the reference count method. It's created to handle regular regions with no references and circular references. When performing a mark and sweep, the program

   (a) Tentatively marks all blocks as useless

   (b) Explores all pointers in the program recursively and marks each block passed as useful

   (c) Walks back through the heap and deletes all blocks that weren't marked as useful

   Since the original exploration stemmed from the program's active pointers, blocks with zero references and circular references unreachable from currently active pointers aren't marked as useful.

3. Stop and Copy The stop and copy follows the same process as the mark and sweep except for the first two steps. Instead, it divides the heap into two halves. The first half will be responsible for all the allocation. When memory runs low, all pointers in the program are explored and each block of memory that's encountered are moved to the second half (pointers are updated with the process), then the notion of the first and second halves are swapped.

1.3.6 Strings

Strings, like arrays, are par for the course. They're arrays of characters that are always one-dimensional, always store one-byte elements, and never contain references.

String operations:

1. Assignment
2. Comparison (=, ¿, etc)
3. Concatenation
4. Substring reference
5. Pattern matching

The "meat" in "memory allocation" comes from how string lengths are determined, i.e. how much memory is allocated and how the program knows the length of the string. Strings are variable in length, so some sort of length descriptor is used to keep track of this information.

1. C uses a null character after the actual string
2. Pascal stores the length of the string in the first character and the string begins at index 1. While this makes it easier to get the length as there's no need to search for a null character, it means that strings are limited to 255 characters (that's how high the space a character is stored in can count)

1.3.7 Sets

Sets are a datatype introduced in Pascal. The actual implementation of the datatype is done via a bit vector, so bitwise operations can be used to compare them.

```
var A, B, C : set of char;
    D, E : set of weekday;

A := B + C; (* union; A := {x | x is in B or x is in C}, bitwise or *)
A := B * C; (* intersection; A := {x | x is in B and x is in C}, bitwise and *)
A := B - C; (* difference; A := {x | x is in B and x is not in C}, bitwise and/not A & !B *)
```

While a set of characters can be stored with 256 bits (32 bytes), a set of integers can take up 500 MB. Pascal sets are limited to 256 values.

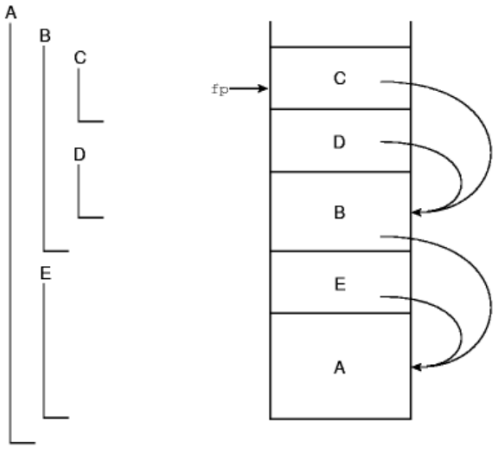
2 Chapter 9: Subroutines and Control Abstraction

2.1 Stack Layout

How can the program access non-local objects in a language with nested subroutines?

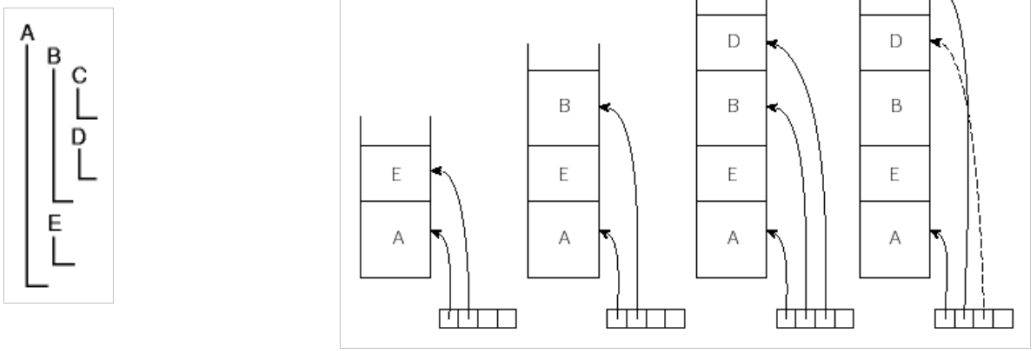
1. Static Chain

   A static chain is composed of static links which are from a subroutine to the lexically surrounding subroutine. To access an object k levels deeper the program must dereference k pointers, which is computationally expensive.



2. Display

   Element j in display references the most recently called subroutine at lexical nesting level j. From a subroutine at lexical level i, to access an object k levels outwards, the program only needs to follow one pointer stored in element $i - k$ in display, which is constant access time.



2.2 Calling Sequences

Maintenance of the stack is the responsibility of the calling sequence:

1. Code executed by caller immediately before and after subroutine call
2. Code executed by subroutine at the beginning (prologue)
3. Code executed by subroutine at the end (epilogue)

Tasks to do around the call:

1. Passing parameters
2. Saving return address
3. Changing program counter
4. Allocate a new frame, change stack pointer
5. Save registers (including frame pointer)

6. Change frame pointer
7. Initialization code for local objects

Tasks to do around the return:

1. Pass return values
2. Finalization code for local objects
3. Deallocate frame, restore stack pointer
4. Restore saved registers (including frame pointer)
5. Restore program counter

2.3 Iterators

An iterator is a control abstraction that allows the enumeration of items of an abstract data type.

1. Traverse an array
   (a) Compute maximum element, average of all elements, display elements, etc.
   (b) Write an enumeration-controlled (for) loop every time
2. Traverse a tree
   (a) Compute maximum node, average of all nodes, count number of nodes, etc.
   (b) Write some (recursive) code to do it

Iterators are similar to `for` loops, but instead of incrementing an `int` that can be used to access data in the data structure being iterated, the loop iterates over the data structure's objects instead. Here's an example of an enumeration controlled loop in Clu.

```
for i in from_to_by(first, last, step) do
    ...
end
```

There's a similar loop in C:

```
foreach(var insect in net.Contents()) {
    insectopedia.Log((Insect)insect);
}
```

Instead of using `var`, the data structure `Insect` can be used as the datatype instead (requiring no casting to work with the current insect).

```
foreach(Insect insect in net.Contents()) {
    insectopedia.Log(insect);
}
```

The overall idea is to iterate over objects instead of using a counter to work with sequential data structures.

2.4 Coroutines

Coroutines execute logic that exists concurrently but only one at a time. The coroutines transfer control to each other explicitly, by name. They're useful for implementing iterators, threads, servers, and discrete event simulation.

Here's an example of a coroutine that scans the drive for corrupted files while displaying a screen saver:

```
    us, cfs : coroutine

    coroutine update_screen
        --initialize
        detach
        loop
            ...
            transfer(cfs)
            ...

    coroutine check_file_system
        --initialize
        detach
        for all files
            ..
            transfer(us)
            ...
                transfer(us)
            ...
            transfer(us)
            ...

begin           --main
    us := new update_screen
    cfs := new check_file_system
    resume(us)
```

For space efficiency, put as much as possible in the callee as tasks in callee appear once in the target program and tasks in caller appear at every point of call.

2.5 Parameter Passing

1. Call by value: the value of actual parameter is copied into formal parameter, the two are independent
2. Call by reference: the address of actual parameter is passed, the formal parameter is an alias for the actual parameter
3. (Ada) Call by value/result: if it's an out or in out parameter - copy formal into actual parameter upon return, change to actual parameter becomes visible only at return
4. (Algol 60, Simula) Call by name: parameters are re-evaluated in the caller's referencing environment, every time they are used, similar to a macro (textual expansion)

2.6 Exception Handling

C++, Ada, Java, ML have a structured approach to handling exceptions: handlers (catch in C++) are lexically bound to blocks of protected code (the code inside a try block in C++).
Exception propagation (if an exception is raised, throw in C++):

1. If the exception is not handled in the current subroutine, return abruptly from subroutine
2. Return abruptly from each subroutine in the dynamic chain of calls, until a handler is found
3. If found, execute the handler, then continue with code after handler
4. If no handler is found until outermost level (main program), terminate program

```
    void foo() {
      try {
          bar();
      } catch (exc) {
```

```
      // handle exception of type exc
  }
}

void bar() {
    baz();
}

void baz() {
  if (/* exception condition met */)
       throw exc();
}
```

3 The Prolog Programming Language

Prolog is a declarative programming language which means it's implicitly written instead of explicitly written. The programmer's task is to write a set of rules that can then be matched to solve the issue at hand instead of writing functions that will get called. The pipeline during Prolog runtime is for the program to attempt to match the current query with the first rule it finds, after which it can run further code in the 'body' of the function conditionally. If criteria in the body is not met, then it continues attempting to match further rules.

```
isSet([]).
isSet([H|T]) :- not(member(H,T)), isSet(T).

isSubset([], _).
isSubset([H|T],S) :- member(H,S), isSubset(T,S).

unionSets([], L, L).
unionSets([H|T], L, L1) :- member(H,L), unionSets(T,L,L1).
unionSets([H|T], L, [H|T1]) :- not(member(H,L)), unionSets(T,L,T1).

intersectionSets([], _, []).
intersectionSets([H|T], L, [H|T1]) :- member(H,L), intersectionSets(T,L,T1).
intersectionSets([H|T], L, L1) :- not(member(H,L)), intersectionSets(T,L,L1).

tally(_, [], 0).
tally(H, [H|T], N) :- tally(H,T,M), N is M + 1.
tally(X,[H|T],N) :- not(X=H), tally(X,T,N).

subst(_, _, [], []).
subst(X,Y,[X|T],[Y|T1]) :- subst(X,Y,T,T1).
subst(X,Y,[H|T],[H|T1]) :- not(X=H), subst(X,Y,T,T1).

insert(X, [], [X]).
insert(X, [H|T], [X,H|T1]) :- X =< H.
insert(X, [H|T], [H|R]) :- X > H, insert(X,T,R).

flattenList([], []).
flattenList([[]|R], L) :- flattenList(R,L).
flattenList([[H|T]|R], L) :- flattenList([H|T],L1), flattenList(R,L2), append(L1, L2, L).
flattenList([H|R], [H|L]) :- flattenList(R, L).
```

4 Final Exam Information

Potential Problems

1. Given some type definitions, specify if the types are equivalent under name equivalence / structural equivalence (See slide 336)

2. Given a program, what does it print with parameter passing by value / reference / value-result / name? (See slide 106)
3. Given a program, what is the content of the display or run-time stack (with its static chain) at some given moment? (See slide 343)
4. Given a C++ program with static / dynamic method binding, what does it print? (See HW3, problem 4)
5. Write a predicate in Prolog (See slide 138)

All slides referenced come from this amalgam presentation