Unit 3 - The Symbol Table & The Environment:

* Be able to define the concept of binding and know the difference between static and dynamic binding
  + Binding is the process of associating an attribute with a name. Names represent language entities or constructs. Attributes are properties that configure the meaning of the name to which they are associated, for example data types, const int x;.
    - Static binding is when the process occurs before execution and is bound statically.
    - Dynamic binding occurs during execution and is bound dynamically.
* Define the symbol table, environment, and memory in the context of binding
  + In the context of binding, the compiler keeps track of the names in a statically scoped, by a data abstraction called a symbol table. A symbol table must support the insertion, look up and deletion of name/identifiers when representing the binding in declarations. In a language with dynamic scoping, an interpreter of the high-level language must initialize and maintain a symbol table at run time.
    - Names -> attributes
  + An environment maintains the bindings of names to locations.
    - Names -> locations
  + Memory in context of binding is storage locations accessed when execution reaches the code to which the declaration is attached.
    - Depending on the language it is managed in different ways
    - Locations -> Value
* Know and define the three principle storage allocation mechanisms
  + Static objects are given an absolute address that remains constant throughout the program’s execution and have read only memory.
  + Stack objects allow you to control how memory is allocated and deallocated in last-in, first-out order, usually in conjunction with returns and subroutine calls. Stack objects come into use if the language permits recursion. Each instance of run time has its own frame and activation record on the stack.
  + Heap objects may be allocated and deallocated at arbitrary times. Heaps are required for dynamically allocated pieces of linked data structures.
    - Ex: linked lists, array lists w/ nodes that can grow dynamically or shrink, lists, sets.
* Know why language using static allocation exclusively cannot support recursion
  + We can never have more than one subroutine call in a recursion function (or any function) with statically allocated objects because they are given a single permanent memory storage location and the recursion would not be able to go back to its original state due to the data from the previous function call was overwritten by the most recent because the function is statically allocated one space in memory.
* Know what an activation record is and what it is comprised of
  + Is also known as a frame in a stack. It contains arguments and return values , local variables, temporaries and bookkeeping information,
    - Think each recursive call of the function gets its own stack frame on the stack to keep of local variables to keep track of the allocated memory
* In terms of space complexity, be able to explain why stack allocation is better than static allocation
  + Stack allocation is better than static allocation because you are able to dynamically create more than one memory space rather than having to have a constant memory space for the whole program
* Know at least two reasons why a heap would be needed for storage allocation
  + A heap is needed for dynamically allocated data,
  + objects whose size may change throughout the program
* Know what a dangling reference is
  + A dangling reference is a reference to an object that no longer exists. Dangling references do not exist in garbage collected languages such as C++ and in java.
    - Ex: have the key but locks have been changed
* Know what garbage is
  + Pieces of memory that have been allocated in the environment but have become inaccessible to the program at some point during the execution.
* Know and be able to define the three types of memory management discussed in class
  + Garbage collection: Garbage collection is the process of automatically reclaiming memory that has been allocated in the environment but has become inaccessible to the program.
  + Manually: example C
  + **Automatic reference counting:** Automatic Reference Counting determines object references at compile time.
* Understand the difference between static and dynamic scoping
  + Be able to determine a programs output using either static or dynamic scoping
  + Be able to generate a syntax table at a specific point in a programs execution using either static or dynamic scoping
    - Static scoped: processed prior to execution
    - Dynamically scoped: processed at execution in a certain path
* Know what a lexical address is and how it is used in static scoping
  + Level and an offset
    - Level is how many nested blocks
    - Offset is the declaration
* Be able to define the concept of a scope hole
  + Scope hole: declaration hidden by another declaration of the same name in one or more nested scopes. Ex: this.get
  + Know what it means when a local variable shadows a global variable
* Be able to define overload resolution
  + This includes what it is and how it works
    - Overloading is of choosing unique function among many with the same name, looks at the data types of the data. It knows which one to take based on the data type and number of parameters to determine which function its calling on.
  + what entities it can apply to
    - can apply: it can work for functions and some allow operator overloading
    - cant
      * ex: void print( int x, int base = 10)
    - arguments with default values are optional and have no way to determine the difference between them
* Know the difference between simple types and complex types
  + Simple types
    - have no other structure than their inherent arithmetic or sequential structure
  + Complex types
    - User can independently define. This type is used when the type has a child element or attribute
  + Understand what a type constructor does and how it relates to basic and complex types
    - **type constructors** can be modeled as operations on sets
    - There are type constructors that do not correspond to mathematical set constructions, and some set operations that do not correspond to type constructors
      * Example: pointer type
      * A set operation not used is intersection
* Know the different simple types discussed in lecture
  + Includes:
  + predefined types
  + ordinal types
  + enumerated types
  + subrange types
* Know the different type constructors discussed in lecture
  + Cartesian product also known as a record:
    - The cartesian product is the cross product of all ordered pairs of elements from two sets. Mathematically iterating all possible combinations. Ability to have everything as one type. Objects or structs that have different properties
    - Allows for related data of different types to be stored and manipulated together
  + Union: In practice, unions have two main use cases:
    - Memory management-storage may be treated as unallocated space, bookkeeping information, user data -The bits are not being reinterpreted, they are being used for independent purposes
    - Two varieties( languages dependent
      * **Discriminated** unions: a **tag** or **discriminator** is added to the union to distinguish the type of its elements
      * **Undiscriminated** unions: lack the tags; assumptions must be made about the type of any value for example in C and C++
    - Unions are useful in reducing memory allocation requirements for structures when different data items are not needed simultaneously
      * Stored in overlapping regions of memory
    - Unions are not needed in object-oriented languages
  + Arrays: Arrays are sometimes called sequence types
    - Arrays have index type and component types. Map an int types to a string type.
  + Pointers:
    - **Reference** or **pointer** constructor: constructs the set of all addresses that refer to a specified type
    - **Reference**: address of an object under control of the system that cannot be used as a value or operated on in any way (except copying)
    - **Pointer**: can be used as a value and manipulated by the programming
    - Example in C:
      * Constructs the type of all addresses where integers are stored
  + Recursive: Recursive type: a type that uses itself in its declaration
* Be able to define type equivalence
  + Way to evaluate if something is equivalent, one way compare sets
* Know the difference between structural equivalence and name equivalence
  + structural equivalence: the constructors are the exact sameGraphical user interface, text, application

    Description automatically generated
  + name equivalence: they are the same if the names are exactly alikeGraphical user interface, text, application, email

    Description automatically generated
* Understand the difference between a statically typed language and a dynamically typed language
  + Statically: data types are checked at compile time.
  + Dynamically: don’t have to specify and can freely change the data type of the variable. Checked at run time
* Define type conversion
  + Type conversion: converting from one type to another
  + The difference between implicit and explicit conversion
    - Implicit: translator does the work, don’t have to cast, translator automatically converts to do arithmetic math b/t types
    - Explicit: Specifically hard code a cast
  + The difference between widening conversion and narrowing conversion
    - **Widening conversion**: target data type can hold all of the converted data without loss of data
    - **Narrowing conversion**: conversion may involve a loss of data
      * **Ex float to int**
* Know the difference between casting and conversion
  + Casting: telling compiler to temporarily treat it differently
  + Conversion permanently changed the data type
* Does casting (as we have defined it) have a purpose in a dynamically typed language?
  + No because there’s no dynamically typed languages don’t do safety checks on data types. therefore no reason to cast

Unit 4 - Common Lisp:

* Know what a cons cell is and what its two components are
  + A cons cell is a pair of values
    - car = value/data of the first item and cdr = memory pointer to the next item. A single cons cell not connected to anything has null cdr
* Know the following about lists:
  + How cons cells are utilized to create a list
    - Con cells are linked together to create lists. The cdr of the first has a memory address to the next cons cell
  + Write a statement which returns a built list
    - (list 1 2 3 4 5)
  + How a list is evaluated
    - Atomic operator, what is it going to do? Function micro or a special operator
    - The other elements of the list get evaluated as literals or variables and pass the results to the function call.
* Explain what the REPL loop is (including what the acronym stands for)
  + Read user input
  + Evaluate input
  + Print evaluation
  + Loop waits for next user input
  + Environment provided by sbcl is a powerful resource for developing and testing code (SBCL for Common lisp)
* Know how to implement selection statements
  + Both if-else and if-else-if design patterns
  + If-else: (if(condition)(true branch)(false branch))
  + If-else-if:
    - (cond ((conditional-1)(branch 1-1))
    - ((conditional-2) (branch 2-1)(branch 2-2) …)
    - ((conditional-3) (branch3-1))
    - (T(branch-4-1)) ; having the last branch be true provides else
    - )
* Be able to define referential transparency and side effects
  + Referential transparency: The value of any function depends on the arguments
  + Side effect: No notion of state since no concept of memory locations with changed values – refereed to as value semantics
* **Remember that functional languages lack assignment**
* Know how to define a function in Common Lisp ++
  + Both a global function
    - (defun name (parameters)
    - (body))
  + anonymous function
    - lambda
      * provides ability to make an anonymous function (create temporary function that does a task and afterwards is gone)
  + Know how functions return values
    - Return the last thing they evaluate
    - Can use special operator return-from.
  + Know the four types of function parameter and how to utilize them in a function’s parameter
    - &required: have to have to it when we use the function
    - &optional: doesn’t necessarily need it but can use it
    - &key: allows the caller to specify keyword parameters to specify which values go with what parameters
    - &rest: variable number as parameters.
* Understand what a special operator is a why they are necessary
  + A special operator are macros in conjunction with each other,
    - Take the IF special operator
    - (if (condition) (true branch) (false branch))
    - IF isn't such a great syntactic construct because the true branch and false branch are each restricted to being a single list
    - This means if you want to perform a sequence of actions in either clause, you need to wrap them in some other syntax.
  + PROGN
    - PROGN takes an arbitrary number of arguments and evaluates the given arguments sequentially
    - PROGN returns the value of the last evaluated argument
  + To abstract away the IF PROGN pattern, Common Lisp provides the WHEN standard macro
  + WHEN takes a condition and an arbitrary number of statements
  + UNLESS which reverses the condition, evaluating the given statements only if the condition is false
* Understand the difference between the variable namespace and the function namespace
  + Variable namespace:
  + Function namespace:
* Be able to define tail recursion
  + Tail recursion: the recursive step is the very last step of the recursive function
* Know how to identify if a function is tail recursive
  + o One way to identify whether a function is tail-recursive is to run the debugger with a break statement in the base case. Then looking at the call stack, you should only see one invocation of the recursive function.
  + o One way without running the function is to look at the recursive call. If there is nothing else to be done (That is if the function doesn’t require the recursive call to return anything.) then that is a good sign that the function is tail-recursive.
  + o The recursive steps are the last steps in any function.
* Know the two benefits of tail recursion relative to standard recursion
  + A
* Know that functions are first class data values / functions as data
  + Be able to implement a function which takes a function as a parameter
  + Be able to implement a function which returns a function as data
* Remember that Common Lisp uses Garbage Collection
* Be familiar with the let, if, cond, defvar, defun, funcall, & lambda structures
  + You will be expected to do both code analysis and basic programming in Common Lisp for the exam