ALGOL

- Java and C built using ALGOL 60
- Simple and concise and elegance
- Universal
- Close as possible to mathematical notation
- Language can describe the algorithms
- Mechanically translatable to machine language

Program Structure

- Hierarchical structure
- Nested Control Structures allows
- Nested Environments

- 2 statement types
  - Declarative
    - Variable declarations
    - Procedure declarations
    - Switch declarations
  - Imperative
    - Computational
    - Control flow of execution

- Major Problems
  - No input/output statements(Assumed people would design their own as hardware I/O varies)
  - Some features too flexible making it hard to understand and implementation difficult
  - Descriptions not easily understandable with 2 versions as both used new notations
- Major Contributions
  - Used another assignment operator ':=' (difference between equivalence and assignment)
  - Block structure that allows for nested scopes (enclosed with BEGIN-END)
  - Parameter Passing

- Replacement of a single statement with a sequence of statements(promotes regularity)

Block

- Each block defines a nested scope
- Variables only visible and exists within it, hidden from outside and gone when out of block
- Avoids variable re-declaration by defining depths of scopes
- Allows larger programs to be written w/o errors and faster
- Common data must be redeclared in each subprogram(repetition of code, voiding abstraction principle)
- Allows efficient stack storage management

- Dynamic Scoping
  - Meaning of statements and expressions evolve during runtime
- Static Scoping
  - Meaning of statements and expressions always stay the same

Syntax

- Still have old control statements(IF-THEN, GOTO)
- FOR-LOOP, WHILE-LOOP, SWITCH statements
- Machine independent
- NO restriction on variable naming
- Concept of keywords(readability and maintenance)

Data Types and Structures

- Int, Real and Boolean
- Lacked a double precision type
- REGULARITY principle the major goal
  - Regular rules w/o exceptions are easier to implement, learn, use and describe
- ZERO ONE INFINITY principle
  - Only reasonable numbers in programming are 0,1 and infinity
- Arrays in ALGOL are generalized(more than 3 dimensions) and dynamic
- Arrays indexed only by integers
- Strong typing and keywords
- Allow recursion
- Parameter passing
  - Pass by value
    - Copy passed, no worry of actual being overwritten
    - Cost expensive with memory (especially with arrays)

- Pass by name
  - Used in recursion
  - Compiler tries to differentiate the same variables by renaming them

Parameter Passing Techniques

- Call by value
- Call by reference
- Call by result
  - Formal parameter acts as an un initialized variables local variable which is given a value during the execution of the procedure
  - On leaving the procedure the value of the formal parameter is assigned to the actual parameter
- Call by value-result
  - Formal parameter affect only the local copy
  - When procedure completes, the actual parameter is updated to the final value of the formal parameter

Static Scoping

- Uses block structure
- Method of binding name to non-local variables
- 2 categories
  - Nested static scopes in subprograms
  - Scopes in sub programs that cannot be nested
- The visibility of the identifiers and the process of binding of names to declarations is determined at compile time
- Allows compiler to perform type-checking
- Easier to read and faster to execute
- Non local access that's works well in many situations
- Problems
  - Can mistakenly call a sub program that should not have been callable(run time error detection)
  - Too much data access(causes incorrect data access/ affects program readability)
  - Getting around the problems can result in the code bearing little resemblances to the original

Dynamic Scoping

- Based on calling a sequence of subprograms not on their spatial relationship, thus the cope can only be determined at run time
- The binding between the use of an identifier and its declaration depends on the execution of the program at run time
- Attributes of non-local variable visible to program cannot be determined statically
- A statement in a sub routine that contains a reference to non-local variable can potentially refer to a different variable entry every time the statement is executed by the program
- Major problems
  - No way to statically type-check references to non-locals
  - Makes it hard to read
  - No protection to local variable access in sub program
  - Take far longer to access non locals

Pointers

- Non existent in fortran nor algol
- 2 uses
  - Power of indirect addressing at higher level than assembly
  - A method of dynamic storage management
- 2 interpretations
  - Reference to contents (normal pointer)
  - Reference to value in memory cell whose address is in the memory to which the variable is bound(dereferencing pointer)
- Dereferencing a pointer is the process of following the pointer to the variable or subprogram whose address it holds
- 2 problems
  - Dangling pointer
    - 2 pointers point to same data
    - Original pointer and data deleted
    - Other pointer points to null
    - Solution is to use a tombstone
      - Pointer points to tombstone that points to data
      - Pointer always points to tombstone even if data gone
      - Cost in memory
      - Cost in processing for keeping track
    - Solution is Lock and Keys approach
      - Pointers represented as ordered pairs (key, address)key is an integer value
      - Heap dynamic variables have a header cell that stores an integer lock value

- When allocated a lock value is created and placed both in lock cell and the heap variable and in the key cell of the pointer
- Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap dynamic variable.
- If match then legal, else its an error
- When deallocated , the lock value is cleared to an illegal lock value
- So if there is another pointer pointing to invalid data, the key value will no longer match and access is denied

    - Memory Leak
        - 2 pointers to 2 data
        - Force the 2 pointers to point to 1 data
        - The other data is now inaccessible as no pointer points to it
        - Solution is reference counter or garbage collection
        - Garbage collector
            - When need most, works worst
            - Uses an algorithm collects heap cell with an indicator bit
            - More efficient by use of the pointer rotation and slide operations
            - 3 phases
                - Cells set themselves to indicate as rubbish
                - Every pointer in the program is traced into the heap and all reachable cells are marked as not being garbage
                - All cells in the heap that have not been specifically marked as still being used are returned to the lost of available space
            - Garbage collection doesn't work all the time
            - Problems (single size cells)
                - Takes too long (cost in execution and memory due to indicator bit)
                - Process yields only a small number of cells as available space
            - Problems(variable size cells)
                - Different sizes, scanning is a problem
                - If pointer doesn't point to anything, cannot follow chain
                - Maintaining list of available space when it becomes a list of variable sized lengths, segments or blocks
                - Allocation slowed because of requests to large block
    - Reference Counter
        - For each memory location, allocate a counter to keep track of the number of pointers associated
        - Problems (single size cells)
            - Cost in memory for counters
            - Cost in updating the counters
            - Does not work with circular linked list
        - Problems (variable size cells)
            - Available space list maintenance problems
- 3 conditions
    - Can change address of pointer
        - Makes pointer point at something else
    - Can change value of the data
    - Cannot change memory location of the data

Weak and Strong Typing

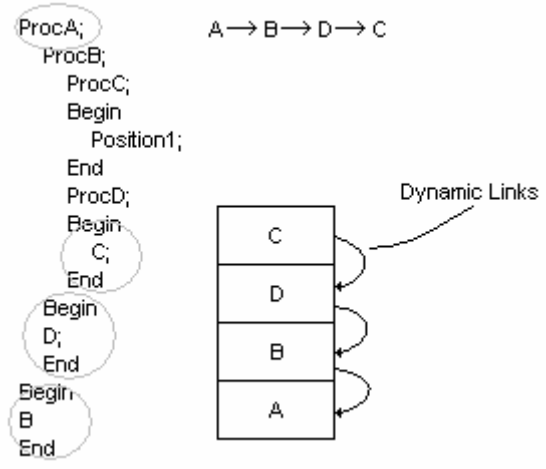
- Strong
    - Eliminate errors before runtime
    - Each name in a program has only a single type that is known at runtime
    - Type of variables known at compile time, allows compiler to catch errors
- Weak
    - Type of variable not known until runtime,
    - Errors undetected by compiler

Activation Record

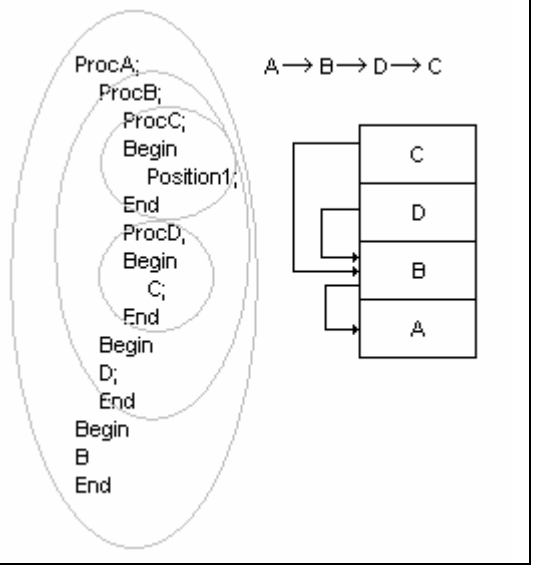
- Need for dynamically created ARs for storing recursive instances
- To know the state of the procedure activation it is necessary to know
    - The code that makes up the body of the procedure
    - The place in the code where the activation of the procedure is now executing
    - The value of the variables visible to this activation
- Code does not vary between different instances therefore not included in AR
- Contains variable parts which define a particular execution
    - Instruction Part
        - Designates the current instruction being executed
    - Environment Part

        - Defines both the local and non-local context to be used for this activation of the sub program – it determines how the instructions are interpreted
        - Local Context
            - Local variabkles, actual parameters and registers
            - Easy to search
        - Non-Local context
            - Difficult to search as it needs to be extracted from the saved states
            - Uses static links
- Dynamic Links
    - o A subprogram needs a reference to its caller
    - o Provide a pointer to the caller's AR record
    - o Pointer is stored in the AR of the subprogram
- Dynamic Chain
    - o Sequence of dynamic links
    - o Tells exactly the order of the subprograms
    - o Search within scope for variable, if not found go to parent up to the ancestor until found else return error.
- Static Link
    - o Points to the bottom of the activation record instance of the static parent
    - o 2 ways to access non-local variables(have to find the definition and the right one)
- Static Chain
    - o Nesting level known at compile time, compiler knows if a variable is non-local and the length of the chain needed to reach it
    - o Static depth indicates how deeply it is nested in the outermost scope
    - o Nesting depth/chain offset is the difference of static depth and to the variable(Ancestor)
    - o Access to a non-local beyond the static parent scope is costly
    - o Steps
        - Find the instance in the AR in which the variable was allocated
        - Use local offset of the variable within the AR
        - Search the static chain until a static ancestor AR has that variable
- Displays
    - o Static links are collected in a single array instead of AR
    - o Contents is a list of addresses of the accessible AR, one for each scope and in order of nesting
    - o Access of non-local only involves 2 steps regardless of scope and reference location of variable
    - o Access to a non-local beyond the static parent scope is costly
    - o All subroutine invoked/ended requires the display to be modified(cost at processing)
    - o A subroutine termination also requires the saved pointer in the AR to be placed back in display
    - o A pointer at N points to an AR record with static depth of N
    - o Disadvantage at memory for implementing an array
    - o Steps
        - Link to correct AR in display is found using a statically computed value called the display offset which is closely related to the chain offset
        - Local offset within the AR is computed and used like in static chain
        - Non-local reference is represented by an ordered pair of integers [display offset | local offset

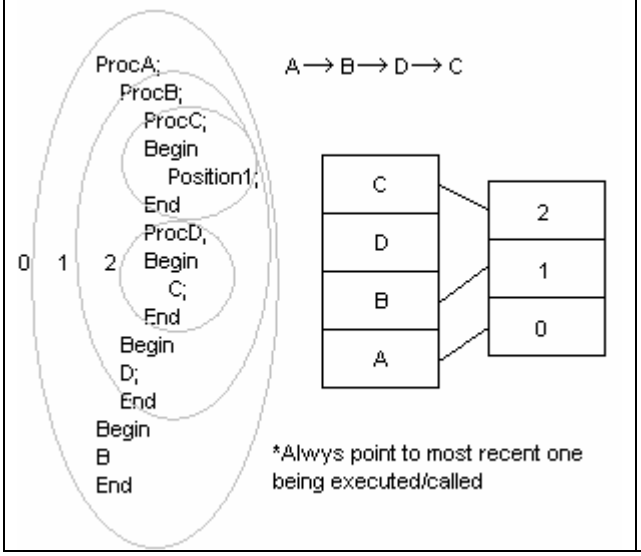
Dynamic Link Chain



Static Link chain



Display



Compund Assignment Operator

- A = A + B
- Same as A += B

Nesting Selectors

- Compiler doesn't bother about indentation can result in improper semantics but compiler doesn't complain
- Use curly braces to indicate proper scope of nesting
- Provides disambiguation