Programming Languages Semantics

The semantics of a program describe its *meaning*. This can be done formally (operational, axiomatic, & denotational semantics; see [Sebesta 96]) or informally.

I. Control Structure Overview
   A. [Bohm & Jacopini 66] showed that all single-entry single-exit control flow patterns can be expressed using nested applications of these three control structures
      1. Sequence
      2. Selection (Conditional) control structure
      3. Iterative (Repetition, or Looping) control structure
   B. The implication is that *goto* is not necessary.
   C. Sequence can also apply *within* a statement
      1. Control within an expression
         a. Precedence, associativity, and parenthesization affect order of actions
            e.g. abs( cos( x + 3 * ++y) / z++)
         b. In C, the operators: &&, ||, and "," are left-associative (i.e. the left-most done first)
            (1). Associativity is undefined for other operators.
            (2). Order is irrelevant unless there are *side effects*
            e.g. func(x) + x
      2. Expression vs. Statement
         a. A *statement* may contain various *expressions*. In contrast to procedural (imperative) languages, pure functional (applicative) language programs consist of a single nested expression.
         b. This implies no side-effects, so the order of evaluation (applying parameter values) does not change the meaning of the program
         c. In Procedural languages, order of evaluation may affect program. Consider C’s *lazy evaluation* that Pascal doesn’t have.
            e.g. while (index < size && array[ index] != key) index++;
II. The _goto_ statement (unconditional branching).

A. Came from Assembler _branch_

B. All other control structures can be rewritten with a selector statement and a goto, however goto's are considered dangerous (Dijkstra's 1966: "Goto considered harmful"). A few languages have been designed without it.

C. Problems with Unconditional Branching

1. There is no syntactic indication of type of control pattern used
2. Makes it difficult to divide program into independent units

D. Restrictions on Branches: In Pascal gotos are allowed but are restricted. The target of a goto can never be in a compound statement of a control structure unless execution of that compound statement has already begun and has not terminated.

1. Examples of goto that are _not_ allowed:

   ```
   while cond1. do
   begin
     while cond2. do
     begin
       100: ...
       ...
       end;
     while cond3. do
     begin
       goto 100;
       goto 200
     while cond3 do
     begin
       200: ...
       ...
       end;
     end;
   end;
   ```

2. Examples of legal goto's in Pascal:

   ```
   while cond1 do
   begin
     100: ...
     ...
     while cond2 do
     begin
       goto 100;
     end;
   end;
   ```

   ```
   procedure sub1;
   label 100;
   ...
   procedure sub2;
   goto 100
   ...
   end;
   ```

E. Example where a goto seems to help the code's readability: (a letter to the editor of CACM by Rubin in 1987). This code finds the first row of an n x n integer matrix that has all zeros.

```
III. Selection Statements: Allows for choosing between execution paths in a program

A. Two-Way Selection Statements

1. Design Issues
   a. What is the form and type of the expression that controls the selection?
   b. Can a single statement, sequence of statements, or a compound statement be selected? (single statement selection leads to gotos.)
   c. How does a "else" match up with previous "if" statements?

2. Single-way Selectors: FORTRAN (logical if) & BASIC
   a. IF (Boolean expr.) statement
      In this FORTRAN selector the statement may not be compound and logical ifs may not be nested. Would need goto, as in:
      
      ```
      IF (.NOT. condition) GO TO 20
      I = 1
      J = 2
      20 CONTINUE
      ```
      Since the code can contain multiple labels, it can contain multiple entry points.
   b. Compound statement in ALGOL 60 adds the capability:
      
      ```
      if (Boolean expr) then
      begin
      statement_1;
      ...
      statement_2;
      end
      ```

3. Examples of Two-Way Selectors
   a. ALGOL 60 was the first, with the form:
      
      ```
      if (Boolean expr) then statement
      else statement
      ```
      Where both statements could be compounded

4. Nesting Selectors: which "if" does an "else" go with? Semantics or Syntax can be used.
   a. Example:
      
      ```
      if (cond1.) then
      if (cond2.) then
        statement1;
      else
        statement2;
      ```
      Which if does the "else" go with?
   b. Static semantics: Language definition tells which it should be. E.g. in Pascal: "else" goes with most recent unpaired "if"
c. Syntax can be used instead: E.g. ALGOL 60 where an if may not be directly nested in a "then" clause, but must be in a compound statement::

```plaintext
if (cond1.) then
  begin
    if (cond2.)
      then statement1
    else statement2
  end

or else as in C & Pascal:

if (cond1.) then
  begin
    if (cond2.) then statement1
  end
else statement2
```

5. Selection Closure: an alternative to the above compound statement for nested "else" is to use special words to mark the end of an if.

a. E.g. in Ada:

```plaintext
if cond1 then
  statement1;
else
  statement2;
end if;

and also:

if cond1 then
  statement1a;
  statement1b;
else
  statement2a;
  statement2b;
end if;
```

Note that inside the "then" and "else" clauses we have statement sequences rather than compound statements. The "end if" marker allows this.

b. The previous nested "else" examples would then look like:

```plaintext
if cond1 then
  if cond2. then
    statement1;
  else
    statement2;
  end if;
end if;

Or as:

if cond1 then
  if cond2. then
    statement1;
  else
    statement2;
end if;
```

6. Functional Languages’ conditional control structure simply returns a value.

E.g. in C: \( z = (x < y ? x : y) \); // Assign minimum of x & y to z
B. Multiple Selection Constructs
- generalization of a selector, & can be built with multiple selectors.
- An additional construct is more clear, however.

1. Design Issues
   a. What is the form and type of the expression controlling selection?
   b. May a single statement, a sequence of statements, or a compound statement be selected?
   c. Is there a "default" clause for unrepresented selector values?

2. Early Multiple Selectors in FORTRAN: three-way selector (arithmetic if)
   a. IF (arithmetic expression) N1, N2, N3
      where the three labels are statements corresponding to the expression values of negative, 0, positive respectively. It would often be used as:
      
      ```
      IF (expression) 10, 20, 30
      10   ...
      ...  
      GO TO 40
      20   ...
      ...  
      GO TO 40
      30   ...
      ...
      40   ...
      ```

      Problem: allows multiple entry & exit points using labels and gotos

3. Modern Multiple Selectors
   a. Pascal:
      ```
      case expression of
         constant_list1: statement1;
         ...  
         constant_listn: statementn;
      end
      ```
      Where expression is of ordinaly type (integer, boolean, character, enumerated type), and selectors are lists.
      (1). Only one of the selectors is executed, constant lists must be mutually exclusive, and the constant lists' union need not be exhaustive. It has single entry & exit point.
      (2). Issue: what about unrepresented selector options? An "else" clause was added to take care of this.

   b. C mutiple selector: without a break, control flow continues to next statement.
      ```
      switch (index) {
      case 1:
      case 3: statement1;
         statement2;
         break;
      case 2:
      case 4: statement3;
         statement4;
         break;
      default: statement5;   //possibly an error message?
      }
IV. Iterative Statements:

Cause a statement or collection of statements to be executed 0, 1, or more times. Thus, sequence is a special case of iteration.

A. Counter-Controlled Loops

1. loop parameters: loop variable, initial value, terminal value, step size

2. Design Issues
   a. What is the type and scope of the loop counting variable?
   b. Is the loop variable defined upon exiting the loop?
   c. Can the loop variable or loop parameters be changed inside the loop?
   d. Are the loop parameters evaluated on every iteration?

3. FORTRAN 77 & 90 DO loop:
   a. loop variable is allowed to be integer, real, or double; loop parameters can be expressions and can have negative values; loop parameters are evaluated at the beginning of the execution to give an iteration count, after which loop parameters may be changed inside the loop without affecting the loop control.
   b. comma optionally added after the label
      ```
      DO 10, K = 1, 10
      SO that "DO 10 K = 1, 10" with an inadvertent decimal replacing the comma would not become the assignment statement:
      DO10K = 1.10
      ```

4. ALGOL 60 for statement: flexibility led to excessive complexity
   a. loop parameters evaluated on each iteration, allowing step size to change at each iteration.
      E.g.
      ```
      i := 1;
      for count := 1 step count until 3 * i do
          i := i + 1
      ```
      Note that the first time through the loop count is assigned one and so stepsize is one. The second time through the loop count has been incremented by stepsize (one) and so is now two. This means that "step count" changes the stepsize to two. Loop does eventually terminate since count increases faster than (3 * i), as it doubles while (3*i) increases by 3 each time.

5. Ada for statement: similar to Pascal
   a. Syntax:
      ```
      for variable in [reverse] discrete_range loop
          ...
      end loop
      ```
   b. Design choices: loop variable is implicitly declared at loop beginning and implicitly undeclared at loop end. E.g.
      ```
      COUNT : FLOAT := 1.35;
      for COUNT in 1..10 loop
          SUM := SUM + COUNT;
      end loop
      ```
      Where the loop "COUNT" masks the outer "COUNT"
6. C & C++ for statement can both count and contain logical tests
   a. Syntax:
      
      ```c
      for (initial_exp.; continuing_cond_expression; post_loop_exp)
        statement
      ```

      Expressions in C can be statements and return a value. Zero value means False, a non-zero value means True.

   b. Example:
      ```c
      for (index = 0; index < 10; index++)
        sum = sum + list[index];
      ```

   c. All expressions in the "for" are optional. It is legal to branch into a "for" loop.

   d. Multiple statements may be used in a single expression in a "for" loop, separated by commas. E.g.
      ```c
      for (count = 0, sum = 0; x > 0; scanf("%d", x), sum += x, count++)
      ```

   e. In C++, the declaration of the loop counter has scope until the end of the function. E.g.
      ```c
      for (int count = 0; count < length; count++) { ... }
      ```

      which is different from block scope only:
      ```c
      { int count;
        for (count = 0; count < length; count++) { ... }
      }
      ```

B. Logically Controlled Loops

1. Design Issues
   a. Is the control pretest or posttest?
   b. Is the loop simply a special case of a counting loop?

2. Examples
   a. C:
      ```c
      scanf( "%d", &indat);
      while (indat >= 0) {
        sum += indat;
        scanf("%d", &indat);
      }
      do {
        indat /= 10;
        digits++;
      } while (indat > 0);
      ```
C. Explicitly exiting loop
   Design issue: should only one loop be exited, or can enclosing loops also be exited?
   1. Ada:

      loop
          ...
          end loop

      which can contain an "exit" statement which may be conditional or unconditional:
      exit [loop_label][when condition]
      For instance:
      loop
          ...
          if SUM >= 1000 then exit;
          {Could also be: exit when SUM >= 1000;}
          ...
          end loop;

      Multiple enclosing loops may be exited using a label:
      OUTER:
          for ROW in 1 .. MAX_ROWS loop
            INNER:
                for COL in 1 .. MAX_COLS loop
                    SUM := SUM + MAT(ROW, COL);
                    exit OUTER when SUM > 1000;
                end loop INNER;
            end loop OUTER;

      2. C: Uses continue to repeat the loop with the next iteration. E.g.
      while (sum < 1000)  {
          scanf("%d", &value);
          if (value < 0) continue; {goto the top of the loop again
          sum += value;
      }

      and also the break statement to exit the loop entirely:
      while (sum < 1000)  {
          scanf("%d", &value);
          if (value < 0) break;   {leave immediate enclosing loop
          sum += value;
      }

D. User-Defined Iteration Control
   - Rather than use a counter, use the number of elements in a user-defined structure. This is
     known as an iterator
     1. LISP: the dolist function iterates on simple lists
     2. Iterator to traverse a tree, example in C:
        for (ptr = root; ptr; traverse( ptr))  {
            ...
        }

V. Subprograms
   A. Purpose: avoid duplication of code
   B. Usually has a header (often using a keyword such as SUB, FUNCTION, PROCEDURE) with
the name and parameter mode

C. Difference between Functions and Procedures.
1. Functions are an expression and can be used along with an assignment. Includes some mechanism to indicate return value
2. Procedures are statements in their own right.

D. Subprogram call & return
1. E.g. a set of invocations: proc1 calls proc2 calls proc3. Proc 3 must exit before returning to proc2. This nesting of invocations is called a hierarchical control structure.
2. System must keep track of local variables’ scope, current instruction being executed in subprogram, values of local variables. This information stored in the activation record, aka a stack frame for that invocation and is stored on the stack. It contains:
   a. local variables
   b. arguments
   c. return address (PC value)
   d. dynamic link: points to the top of an instance of the activation record of the caller.
      This is used in the destruction of the current activation record instance when the subprogram is completed, as there may have been other allocations on the stack.
   e. static link: points to the bottom of the activation record instance of an activation of the static parent. It is used to access non-local variables
   f. register contents (e.g. PSW, AC)
3. If proc1 calls proc2 calls proc3, we get:

A procedure can only access its variables and other non-local variables through its static link (discussed further later)

E. Recursion: Can only be fully understood by using the stack.
1. Consider the factorial function:

```c
long factorial( long n)
{
    if (n == 1)
        return 1;
    else
        return n * factorial( n - 1);
}
```

which being called with "factorial( 3)" creates the stack:

```
factorial a; PC = 2; n = 1
factorial b; PC = 4; n = 2
factorial a; PC = 4; n = 3
```

Note that each recursive call has its own activation record containing the current value for n as well as the next line to be executed once it continues.
2. Note that older versions of FORTRAN, COBOL, & BASIC would statically allocate space for only one version of all variables, and so did not support recursion since there was no run-time stack.
3. Advantages of recursive approach: code is simpler, don’t need loop counter variables
4. Disadvantages of recursive approach: takes more memory to store stack frames

VI. Parameter-Passing Methods

A. Parameters & Scope
   1. *Statically-typed* language: type of parms. must be predefined and specified in subprog. definition
   2. *Strongly-typed* language: type of argument must be same as type of parameter
   3. We can have *local variables* whose scope is the subprogram only

B. *Formal vs. Actual* parameters:

```c
double square( int x)
{
    return x * x;
}
```

C. Semantics models of Parameter Passing
   1. Modes:
      a. In mode: formal parameter used to receive data from the actual parameter
      b. Out mode: formal parameter ignores corresponding actual parameter value, used to return data into the actual parameter
      c. Inout mode: both of the above
   2. Data transfer involves either
      a. copying a value (data transfer)
      b. transmitting an access path (address, or pointer)

D. Implementation Models of Parameter Passing
   1. Pass-by-Value
      a. The value of the actual parameter is used to initialize the formal parameter, which then acts as a local variable (in mode).
      b. This is normally implemented by data transfer
      c. Pass-by-value is the default in C
      d. Main disadvantage: additional storage is required, which can be expensive for large objects (may want to pass it as a variable parameter even if it won’t be changed...)
   2. Pass-by-Reference
      a. Transmits an access path, is inout-mode.
      b. Is efficient in terms of space: no local copy need be made, though indirect addressing must be used.
      c. Avoids inefficiency of pass-by-value when passing a large object, only now it allows the subprogram to modify that object. For this reason C has pass by *constant reference*. Ada compilers do this automatically when the passed object is large enough.
d. Compare C’s pass by reference to C++ pass by reference:

```c
/* In C */
int double( int *num)
{
    return *num * 2;
}

int main()
{
    int y = 3;
    int x;
    x = double( &y);
}
```

```c++
/* In C++ */
int double( int &num)
{
    return num * 2;
}

int main()
{
    int y = 3;
    int x;
    x = double( y);
}
```

3. Pass-by-Name
   a. Does textual substitution for the formal parameter in all its occurrences in the subprogram. It is like a run-time macro substitution
   b. It is flexible, though difficult to understand
   c. Example in pseudo-Algol

```c
PROCEDURE bigsub;
INTEGER global; INTEGER ARRAY list [1:2];
PROCEDURE sub(parameter);
INTEGER parameter;
BEGIN
    parameter := 3;
    global := global + 1;
    parameter := 5;
END;
BEGIN
    list[1]:= 2; list[2]:= 2; global:= 1;
    sub( list[ global]);
end;
```

After execution, list[1] and list[2] have the values 3 and 5, respectively. This is an example of late binding which is used in OOP
VII. Name Structure
   A. Identifiers
      1. Naming entities
         a. Program entities given a name called an *identifier*
         - can refer to variable, constant, subprogram, parameter, type, or object
         b. An identifier is bound to a *referent*, which is the actual entity
         c. Identifier-to-referent correspondence is not one-to-one;
            e.g. int x can be different values
            In Functional languages: ea. name unique & can’t be rebound
         d. Identifiers should be mnemonic
      2. Reserved words: language-defined names not available as identifiers (e.g. BEGIN)
      3. Binding
         a. Binding couples not only identifier w/referent, but may associate type or correct
            fcn. call w/overloading (e.g. "+" for reals or ints)
         b. Current *environment* is set of bindings in effect
         c. *Static* binding: made before execution
            *Dynamic* binding: determined at run time or during execution
         d. Compile-time binding: Absolute code [See overhead]
            Load-time binding: Relocatable code
            Run-time binding: Dynamic code
   B. Declarations
      1. Declaration is non-executable statement introducing an identifier into some prog. scope
         - Statically typed lang.: also give *type* of referent
         - C++ allows declarations at any point
      2. Declaration vs. *definition*
         Definition instantiates the identifier, binding name w/the entity.
      3. *Forward* declarations, or *prototypes*: Using an identifier that is defined later in program
         Used with mutually referential subprograms, types pointing to each other.
         E.g.1: C function declaration before main, when body of function is after main
         E.g. 2: Function is external [see overhead]
      4. Symbol table: stores attributes & bindings of program identifiers
         a. Stores location: stack frame offset for local variable
         b. Symbol table exists only during compilation for compiled lang.
            Must exist during execution for interpreted languages
      5. Typing
         a. *Static typing*: type of object is in symbol table
            Type of subprogram is *argument signature*
         b. *Dynamic typing*: identifier may be bound to different types during execution
            - E.g. object x could refer to an int at one point, and a real at another
            - Here we have *dynamic binding* with *polymorphic* identifiers (can refer to
              objects of different types)
            - Type info. must be stored w/ea. object
C. Scope
Determines extent in which an identifier has meaning
1. Block-structured language: block scopes can be nested
2. Nesting allows for top-down design (a.k.a. stepwise refinement, functional decomposition)
3. C: subprograms cannot be nested, but data declarations can (inside braces)
   e.g. if (cond) { int temp; stmnt1; stmnt2;}
4. Identifier declarations
   a. Identifiers can be local, nonlocal, global
   b. Redefinition in an enclosed block hides the external version of same identifier
   c. Identifier resolution decides which one to use
   d. A qualified name allows specifying both scope and name
      e.g. scope.name in ADA, and scope::name in C++
5. Lexical scope
   a. Static scope: names bound when compiled
   b. Dynamic scope: names bound when run
   E.g. [overhead of Pascal scope]
      Identifier resolution can see outward, but not into a scope
      Enumerate identifiers available in each scope
D. Mechanism for Implementing Nonlocal References
1. Background:
   a. A two-step process:
      (1). Find the instance of the activation record
      (2). Use the local offset of the variable within the activation record instance.
   b. Note that a procedure is callable only when all of its static ancestor program units are active.
   c. As in the case of recursion, the parent scope activation record need not be adjacent to the child's activation record.
2. Static Chain: a chain of static links connecting certain activation record instances in the stack
   a. During execution, the static link of an ARI (Activation Record Instance) P points to an ARI of P's static parent unit. This continues for the parent, etc., giving a static chain connecting all the static ancestors of P.
   b. At compile time the compiler can determine not only the length of the chain needed to access a nonlocal variable, but the offset within the ARI as well. This is done as follows:
      (1). Static depth is the depth of nesting from the outermost scope, with the main program being depth 0.
      (2). The static distance (chain_offset) between a calling procedure and the called variable is the difference between the static depth of the calling procedure and the static depth of the procedure containing the declaration for the called variable.
      (3). The local offset within an ARI can be determined at compile time as well, so actual variable names need not be stored in the ARI
      (4). These concepts are illustrated in the [overhead] and in the text
      (5). Note that the (chain,local) offsets are determined statically, as the compiler keeps track of nesting information to verify scopes as compilation progresses.
c. Returning from a subprogram means simply removing the subprogram’s ARI from the stack
d. Action performed at subprogram call: compiler determines nesting depth between caller and enclosing scope of subprogram being called. This nesting depth is the length of the static chain to follow to set the static link of the new ARI for the subprogram being called. [overhead, where Sub3 calls Sub1]
e. Problem: references to variables beyond the static parent are costly, since a potentially lengthy chain may need to be followed.

E. Programmer-defined visibility
   1. Problems with use of nonlocal identifiers
      Using a non-local identifier means a module is no longer self-contained; This is essentially a side effect.
   2. Limitations of hierarchical scope
      a. Consider trying to share the following two arrays among the 3 procedures shown below:

      This can not be done in a hierarchically scoped language such as Pascal.
      b. In C, to avoid linker errors caused by multiple includes of the same thing, use:

      Note that the dependencies shows globals.h included in through two paths, which would cause linker errors unless #ifndef is used as shown above.

   3. Exporting, importing and hiding names
      A mechanism is needed to explicitly specify scope. For this reason the name space of a module is separated into:
      a. Interface: visible to outside world, can be imported by others that want to use it (e.g. #include in C)
      b. Implementation: private to the module, contains local declarations that are visible locally only, as well as the body of the elements visible to the outside world that are listed in the interface.

F. Overloading
   1. More than one function can have the same name (function overloading), where the argument signature is used for unique identification
   2. Operator overloading is just a special case of function overloading
      e.g. 5 + 3 = +(5,3) [see overhead]
VIII. Data Structure
A. Data Objects

1. Purpose & Definition: information that must be represented by the application
   Differentiate between:
   a. Program Object: Program’s logical named entities; names are identifiers, entities are types
   b. Storage Object: How the program object is actually stored in memory; An instantiation of the logical entity

2. Variables: association between name & data object [See Fig. 0.5]
   a. l-value: l.h.s. of assignment: address of storage location
      Used for passing parameters by reference
   b. r-value: r.h.s. of assignment: results in a value
      Used for passing parameters by value

3. Constants: symbolic name rather than literal value
   a. Pascal: symbolic constants (e.g. CONST array_limit = 10)
   b. C: pre-processor macro (e.g. #define array_limit 10)
   c. Read-only variables (C++, ANSI C) Is constant after initialized; can be initialized using argument values in a subprogram
   d. const function parameter (C++, ANSI C). Allows passing reference parameter which can not be changed by function. const can be used with pointers:
      (1). char *const ptr; /* ptr cannot change, but data can */
      (2). const char *ptr; /* ptr can change, but data cannot */
   e. passing an array is essentially case #1 above, pass by constant pointer

4. Lifetime of an Object
   a. Variable definitions or dynamic allocation (e.g. malloc, new) instantiate the data type (declaration) and allocate storage (definition) [dec.. before def...]
   b. Lifetime != scope: when in subprogram, variable may not be accessible, but still exists

5. Initialization and Assignment
   - Initialization stores a meaningful value immediately after creation (none in Pascal)
   - assignment replaces current value with a new value

B. Storage Allocation Policies
Allocation policies can be static, automatic, or dynamic

1. Memory Management
   If allocation is automatic or dynamic, memory management must be provided [Fig. 0.6]

2. Static allocation
   Objects allocated when program begins and remain until it ends. No Recursion
   Can have static vars. in C

3. Automatic stack allocation: Binding objects to local variables and arguments of subprogram or block
   a. Nested lifetimes
   b. Allows recursion

4. Dynamic heap allocation: allows for programmer-controlled run-time allocation of variable amounts of memory (e.g. for trees, lists, graphs; using malloc and new).
   - This is known as the heap or free store
   - Memory management system must have:
      a. compaction to handle fragmentation
      b. free space allocation algorithm: best fit, next fit, worst fit, buddy system
5. Deallocation of dynamic storage
   a. **dispose** and **free**
   b. **memory leak**: programmer forgets to release storage
   c. **dangling pointer**: referring to previously released storage
   d. **automatic storage reclamation**: run-time recovery of unreferenceable storage
      - may use **reference counting**
   e. Recovery of cyclically linked unreferenceable storage known as **garbage collection**.

C. Data Types
   1. Purpose and Definition
      a. To completely define a type, must have:
         (1). What it **is**: Structure of its instances (all you get in C or Pascal)
            a.k.a. representation of legal values
         (2). What it **does**: Valid Operations
      b. Values & operations should correspond to states & behaviours of domain entities: using an array should **look** like an array
   2. Types and Instances
      a. In compiled languages, type does not create an object, only a name
      b. A static object is represented in the disk image of the executable, but dynamic objects are created only during execution
         Eg. In Pascal [See example] Point type vs. instances
   3. Primitive and built-in types
      Examples: INTEGER, REAL, DOUBLE, LIST, STRING
   4. Numeric Types
      - Would like the **logical** declaration to be separate from underlying **physical** representation. (e.g. declare a whole number that can be *any* size)
      - Ada allows specifying the precision as well as the range
         a. short & long ints
         b. single and double precision reals
         c. bit-strings
         d. positive & negative values
   5. Logical Values (boolean)
      none in C (uses int)
   6. Text data
      - characters
      STRINGS: **counted string** or **terminated string**

D. Defining Types
   1. Development of the type construct
      a. Type was recognized as existing separately from instances
         E.g. ADT is separate from implementation. The operations allowed on a binary tree would stay the same even if the underlying implementation were changed from an array to a true tree.
      b. OOP went a step further in allowing **inheritance** among type definitions
2. First-class objects: Objects that have complete manipulations available. An object is first class if it can be used anywhere a primitive object can be used.
   a. Predefined types \textit{int} \& \textit{real} in C are 1st class
   b. PASCAL \textit{type} describes structure, but it is not 1st class: it doesn’t specify operations, can’t be return type of function
   c. Examples of manipulations: initializations, assignments, equality comparisons, available as parameter, return type, base type, I/O operations, arithmetic operators

3. Type declarations
   a. Tells program "Here is a type name" but neglects the structure, which is given elsewhere. E.g.:
      \begin{verbatim}
      typedef struct node NodeStruct;
      struct node {
        char word[15];
        NodeStruct *nextPtr;
      };
      \end{verbatim}
     Creates symbol table entry for \textit{struct node}, but doesn’t specify structure
   b. Declaration requires some mechanism to combine other pre-existing types (type constructor), listing structure and possibly limiting valid values (e.g. subranges)

4. Type definitions and the type object
   Type definition gives: \textit{name}, \textit{kind} (e.g. array, record), \textit{structure}, \textit{operations}, (also the hierarchy in OOP)
   This creates additional symbol table entry information, but still doesn’t allocate storage

5. Type operations
   a. Could define a generic function which will be used variously, depending on callers type. e.g. generic \textit{sort} function that operates on any type. (C++ templates)
   b. C type operations: \texttt{sizeof}, also numeric type limits in \texttt{limits.h}
   c. OOP types called \textit{classes}. e.g. operations called \textit{methods} in Smalltalk

E. Programmer-defined types
1. Type aliases
   e.g. in Pascal: TYPE Length = REAL;
   - However new & existing types are same
   - ADA derived type:
     type LENGTH is new FLOAT;
     type AREA is new FLOAT
     Derived type LENGTH is same as base type FLOAT (same functions & operations), but is semantically a different type (e.g. an instance of AREA can’t be added to an instance of LENGTH)

2. Pointers
   a. pointer is an "l-value": is a reference to object (address), not the value of object
   b. "No value" is \textit{nil} in Pascal, \texttt{NULL} in C
   c. dereference & indirection:
      -postfix ^ in PASCAL, prefix * in C
d. Difference between object and pointer assignment

```c
/* Object and Pointer assignment in C */
int main()
{
    int *ptr1;
    int *ptr2;
    ...
    *ptr1 = *ptr2; /*Objects now have same value*/
    ptr1 = ptr2;  /*Both pointers now point to ptr2's referent*/
}
```

In C, the & operator gives the address of an object
e. void * is what is returned by malloc, as in:
   ptrToDoubleArray = (double *) malloc( sizeof( double) * arrayLimit);
f. Unfortunately no range checking is done on pointer operations in C. Arrays passed as address of first element only

3. Enumerated Types

   Pascal: TYPE Day = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat); {is 1st class}
   C: enum Day {Sun, Mon, Tues, Wed, Thurs, Fri, Sat};
   a. Values typically mapped onto integers internally
   b. ADA: overloaded enumeration values
      type SCREEN_COLOR is (RED, GREEN, BLUE);
      type TRAFFIC_LIGHT is (RED, YELLOW, GREEN);
      type of variable RED used to determine which of the two REDs is intended. If either could be used, a cast is needed, e.g. SCREEN_COLOR'(RED)

4. Constrained types

   Subrange of existing type may be defined, e.g. in Pascal:
   TYPE UpperCase = 'A'..'Z';

5. Arrays: used to process collections of like objects.
   a. Defined by mapping from contiguous range of integers to a set of elements of the base type
   b. bounds may be fixed (e.g. lower bound = 0 in C) or programmer defined (in PASCAL)
   c. allocation:
      Static: FORTRAN
      Automatic: PASCAL (e.g. declared in a subprogram)
      Dynamic: C. E.g.
      ptrToArray = (int *) malloc( sizeof( int) * MAXSIZE);
   d. Subscription is a primary operation and is shorthand for:
      (1). retrieve(Array, Index): Element
      (2). store( Array, Index, Element)
   e. In ADA an "unconstrained" array type can be used to pass arrays of different sizes [See overhead]
   f. In C: since arrays are passed as address to first element, and arrays cannot be assigned or compared, there is essentially no array type! [except in your head]
   g. Initializing array
      e.g. in C int arr[5] = {1,2,3,5,8};
6. Records
   Composite object made up of collection of fields, where each field has name & type.
   Record instance has a value for each field
   a. Can have default values assigned (ADA)
   b. Field selection operator is usually a period
      E.g. node^.next {In PASCAL}, or array[5].salary

7. Unions
   Allows representing different types at different points in execution
   a. E.g. variant record in PASCAL [See overhead]
      Problem: Can define as one type, initialize, change type & retrieve as different type
   b. ADA solves this problem of type safety for variant records [See overhead]
   c. Undiscriminating union in C [See overhead]
      Space allowed for max. size; programmer must keep track of which of the three types is used, which usually means the union is inside a struct which contains a type flag of some kind.

8. Abstract Data Types (ADT)
   a. Programs that don’t support this (C & PASCAL) allow programmer access to fields in a record via component selection
   b. ADT: definition & operations are in a separate package. These operations must be called to access component values.

9. Subprogram types
   Can pass a subprogram as an argument in another subprogram.
   [E.g. PASCAL on overhead]

F. The semantics of types
   Type specifies a domain over which its operations apply
   1. Type checking
      a. Strongly typed (e.g. PASCAL) compiler enforces domain identity of objects and domain requirements for operators
      b. Weakly typed (e.g. Pre-ANSI C)
         Type checking not always enforced or can be circumvented. E.g. sqrt() expects a double, but could be passed an int [sqrt(7)] which would be treated as a double
      c. Dynamically typed: type checking done at execution time
   2. Explicit casts and conversions
      promotion: (float) integer_variable;
      demotion: (int) real_variable;
   3. Implicit coercions
      Cast or conversion performed implicitly: real_var = int_var * 3.14;

IX. Summary and Review
   A. Control Structure
   B. Name Structure
   C. Data Structure