Data Types

- most programming languages include a notion of types for expressions, variables, and values
- two main purposes of types
  - imply a context for operations
    - programmer does not have to state explicitly
    - example: a + b means mathematical addition if a and b are integers
  - limit the set of operations that can be performed
    - prevents the programmer from making mistakes
    - type checking cannot prevent all meaningless operations
    - it catches enough of them to be useful

Type Systems

- bits in memory can be interpreted in different ways
  - instructions
  - addresses
  - characters
  - integers, floats, etc.
- bits themselves are untyped, but high-level languages associate types with values
  - useful for operator context
  - allows error checking

Type Checking

- type checking ensures a program obeys the language’s type compatibility rules
  - type clash: violation of type rules
- a language is strongly typed if it prohibits an operation from performing on an object it does not support
- a language is statically typed if it is strongly typed and type checking can be performed at compile time
  - few languages fall completely in this category
  - questions about its value

Types Systems

- a type system consists of
  - a way to define types and associate them with language constructs
    - constructs that must have values include constants, variables, record fields, parameters, literal constants, subroutines, and complex expressions containing these
  - rules for type equivalence, type compatibility, and type inference
    - type equivalence: when the types of two values are the same
    - type compatibility: when a value of a given type can be used in a particular context
    - type inference: type of an expression based on the types of its parts

Type Checking

- examples
  - Common Lisp is strongly typed, but not statically typed
  - Ada is statically typed
  - Pascal is almost statically typed
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically
**Type Checking**
- a language is dynamically typed if type checking is performed at run time
  - late binding
  - List and Smalltalk
  - scripting languages, such as Python and Ruby

**Polymorphism**
- polymorphism results when the compiler finds that it doesn't need to know certain things
  - allows a single body of code to work with objects of multiple types
  - with dynamic typing, arbitrary operations can be applied to arbitrary objects
    - implicit parametric polymorphism: types can be thought of to be implied unspecified parameters
    - incurs significant run-time costs
    - for example, ML’s type inference/unification scheme

**Data Types**
- specified in different ways
  - Fortran: spelling of variable names
  - Lisp: infer types at compile time
  - List, Smalltalk: track at run time
  - C, C++: declared at compile time

**Classification of Types**
- most languages provide a set of built-in types
  - integer
  - boolean
    - often single character with 1 as true and 0 as false
    - C: no explicit boolean; 0 is false, anything else is true
  - char
    - traditionally one byte
    - ASCII, Unicode
  - real
Classification of Types

• numeric types
  – C and Fortran distinguish between different lengths of integers
  – C, C++, C#, Modula-2: signed and unsigned integers
  – differences in real precision cause unwanted behavior across machine platforms
  – some languages provide complex, rational, or decimal types

• alternatively, can be declared as constants
  ```c
  const sun = 0; mon = 1; tue = 2; wed = 3; thu = 4; fri = 5; sat = 6;
  ```
  ```c
  enum weekday = {sun, mon, tue, wed, thu, fri, sat};
  ```

• subrange types
  – contiguous subset of discrete base type
  ```c
  type water_temperature = 273.15..373; (* degrees Kelvin *)
  ```
  – helps to document code
  – most store the actual values rather than ordinal locations

• composite types
  – records
    • fields of possibly different type
    • similar to tuples
  – variant records (unions)
    • multiple field types, but only one valid at a given time
  – arrays
    • aggregate of same type
    • strings

• enumeration types
  – facilitate readability
  – allow compiler to catch errors

• Pascal example
  ```pascal
  type weekday = (sun, mon, tue, wed, thu, fri, sat);
  ```
  – orders, so comparisons OK
  ```pascal
  for today := mon to fri do begin ...
  ```
  ```pascal
  var daily_attendance: array [weekday] of integer;
  ```

• composite types (cont.)
  – sets
    • discrete
    • like enumerations and subranges
  – pointers
    • good for recursive data types
  – lists
    • head element and following elements
    • variable length, no indexing
  – files
    • mass storage, outside program memory
    • linked to physical devices (e.g., sequential access)
Orthogonality

- Orthogonality is a useful goal in the design of a language, particularly its type system. A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined (analogy to vectors).

For example
- Pascal is more orthogonal than Fortran, because it allows arrays of anything, for instance, but it does not permit variant records as arbitrary fields of other records (for instance).
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about.

Type Checking

- A type system has rules for:
  - Type equivalence (when are the types of two values the same?)
  - Type compatibility (when can a value of type A be used in a context that expects type B?)
  - Type inference (what is the type of an expression, given the types of the operands?)

- Type compatibility / type equivalence:
  - Compatibility is the more useful concept, because it tells you what you can do.
  - Terms are often used interchangeably (incorrectly).

- Formatting does not matter:

  ```
  struct { int a, b; }
  ```

  Is the same as:

  ```
  struct {
    int a, b;
  }
  ```

- Want them to be the same as:

  ```
  struct {
    int a;
    int b;
  }
  ```

- Two major approaches: structural equivalence and name equivalence:
  - Name equivalence is based on declarations.
  - Structural equivalence is based on some notion of meaning behind those declarations.
  - Name equivalence is more fashionable these days.
at least two common variants on name equivalence
– differences between all these approaches boils down to where the line is drawn between important and unimportant differences between type descriptions
– in all three schemes, we begin by putting every type description in a standard form that takes care of "obviously unimportant" distinctions like those above

structural equivalence depends on simple comparison of type descriptions
– substitute out all names
– expand all the way to built-in types
• original types are equivalent if the expanded type descriptions are the same

example in Ada illustrating type conversion

```ada
a : integer;  -- assume 32 bits
b : real;     -- assume IEEE double-precision
t : test_score; -- as in Example 7.8 (int..1.5000)
c : colons_temp; -- as in Example 7.10 (integer)
...
```

coercion
– when an expression of one type is used in a context where a different type is expected, one normally gets a type error
– but what about
```ada
var a : integer; b, c : real;
...
c := a + b;
```

coercion
– many languages allow such statements, and coerce an expression to be of the proper type
– coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
– Fortran has lots of coercion, all based on operand type

C has lots of coercion, too, but with simpler rules:
– all floats in expressions become doubles
– short int and char become int in expressions
– if necessary, precision is removed when assigning into LHS

coercion
– when an expression of one type is used in a context where a different type is expected, one normally gets a type error
– but what about
```ada
var a : integer; b, c : real;
...
c := a + b;
```
Type Checking

- example in C, which does a bit of coercion
  ```
  short int e;
  unsigned long int l;
  char c;  /* may be signed or unsigned -- implementation-dependent */
  float f;  /* usually IEEE single-precision */
  double d;  /* usually IEEE double-precision */
  ...
  e = l;  /* l's low-order bits are interpreted as a signed number. */
  l = e;  /* e is sign-extended to the longer length, then
          its bits are interpreted as an unsigned number. */
  e = f;  /* e is either sign-extended or zero-extended to e's length, the
          result is then interpreted as a signed number. */
  f = l;  /* l is converted to floating-point. Since f has fewer
          significant bits, some precision may be lost. */
  d = f;  /* f is converted to the longer format; no precision lost. */
  f = d;  /* d is converted to the shorter format; precision may be lost.
          If f's value cannot be represented in single-precision, the
          result is undefined, but NOT a dynamic semantic error. */
  ```

- make sure you understand the difference between
  - type conversions (explicit)
  - type coercions (implicit)
  - sometimes the word 'cast' is used for conversions (e.g., C)

Type Checking

- in effect, coercion rules are a relaxation of type checking
  - recent thought is that this is probably a bad idea
  - languages such as Modula-2 and Ada do not permit coercions
  - C++, however, goes hog-wild with them
  - they're one of the hardest parts of the language to understand

Records (Structures) and Variants (Unions)

- records
  - allow related heterogeneous types to be stored together
  - C, C++, and Common Lisp: structure
    - C++: class with globally visible members
  - Fortran: types

Records (Structures) and Variants (Unions)

- universal reference types
  - a reference that can point to anything
    - C, C+: void
    - Chu: any
    - Modula2: address
    - Java: Object
    - C#: object

Records (Structures) and Variants (Unions)

- C
  ```
  struct element {
    char name[255];
    int start, member;
    double score, weight;
    . . .
  }
  ```

- Pascal
  ```
  type ten-char = packed array [1..15] of char;
  type record = record
    name : ten-char;
    atomic_number : integer;
    atomic.weight : real;
    metallic : Boolean
  end;
  ```
Records (Structures) and Variants (Unions)

• each record component is known as a field
  – most languages use dot notation to refer to fields

  - Fortran: copper%name

• both reference
  – malachite.element_yielded.atomic_number

• in C, struct tags appear in struct definitions as

  ```c
  struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    bool metallic;
  };
  ```

• in C, need typedef
  • typedef struct ...

• in C++, struct tag is type name

Records (Structures) and Variants (Unions)

• records
  – example
  – holes for alignment

  ![Diagram showing a record layout with fields name, atomic_number, atomic_weight, and metallic]

• records
  – usually laid out contiguously
    • addresses are computed as base + offset
  – possible holes for alignment reasons
  – smart compilers may re-arrange fields to minimize holes (C compilers promise not to)
  – implementation problems are caused by records containing dynamic arrays

Records (Structures) and Variants (Unions)

• records
  – some languages, such as Pascal, allow the data to be packed
    • compiler should optimize for space over speed

  ```pascal
  type element = packed record
    name : text;
    atomic_number : integer;
    atomic_weight : real;
    metallic : boolean;
  end;
  ```
Records (Structures) and Variants (Unions)

- records
  - packed layout
    
    | name         | atomic, |
    |--------------|---------|
    | number       |         |
    | atomic_weight| metallic|

  - less space, but system takes longer to access elements

- records
  - record equality can be done field by field
  - to save time, a block compare function can be used
    - compares bytes
    - problems?
      - holes need to be filled with 0's to allow block comparisons to work correctly

Records (Structures) and Variants (Unions)

- unions (variant records)
  - space was extremely limited in early days of programming
  - overlay space
  - cause problems for type checking
  - can be used for automatically re-interpreting bits without casts

Records (Structures) and Variants (Unions)

- records
  - some languages, such as Ada, allow record assignment
    - my_element := copper
  - or record tests for equality
    - if my_element = copper then
  - most other languages do not

Records (Structures) and Variants (Unions)

- records
  - holes in records waste space
    - packing helps, but with performance penalties
    - smart compilers rearrange fields automatically (not C compilers)
    - programmer unaware
    - some systems-level applications may depend on non-rearranged fields

Records (Structures) and Variants (Unions)

- Pascal
  - type long_string = packed array [1..2000] of char;
  - type string_ptr = ^ long_string;
  - type element = record
    - name : two_char;
    - atomic_number : integer;
    - atomic_weight : real;
    - metallic : boolean;
    - case naturally occurring : boolean;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
    - atomic_number := 59;
    - atomic_weight := 121.6;
    - metallic := true;
    - name := "copper";
Records (Structures) and Variants (Unions)

- variant part must be at end

<table>
<thead>
<tr>
<th>atomic number</th>
<th>atomic weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>metallic</td>
<td>source</td>
</tr>
<tr>
<td>prevalence</td>
<td></td>
</tr>
</tbody>
</table>

Arrays

- arrays are the most common and important composite data types
- unlike records, which group related fields of disparate types, arrays are usually homogeneous
- semantically, they can be thought of as a mapping from an index type to a component or element type

Syntax and operation
- index usually inside square brackets [ ]
- some languages use parentheses ( )

Records (Structures) and Variants (Unions)

- Modula-2: variant part not required to be at end

<table>
<thead>
<tr>
<th>source</th>
<th>lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_number</td>
<td>atomic_weight</td>
</tr>
</tbody>
</table>

Arrays

- index
  - most languages require the index to be an integer
  - or at least discrete type
  - non-discrete type
    - associative array
    - Python: dictionary
    - require hash table for lookup

Arrays

- declaration
  - C
    ```c
    char upper[20];
    ```
  - Fortran
    ```fortran
    character, dimension (1:20) :: upper
    char (20) upper    
    ```
  - Pascal
    ```pascal
    var upper : array [0..9] of char;
    ```
  - Ada
    ```ada
    upper : array (character range 'a'..'z') of character;
    ```
Arrays

• multidimensional arrays
  - can be accessed as matrix [3][4]
  - sometimes as matrix [3, 4]
  - Ada: matrix (3, 4)

• slicing
  - some languages allow operations on arrays
    - Ada: lexicographic ordering: if A < B
    - Fortran: more than 60 intrinsic functions
      - arithmetic operations, logic functions, bit manipulation, trigonometric functions

Arrays

• a slice or section is a rectangular portion of an array

Arrays

• dimensions, bounds, and allocation
  - **global lifetime, static shape** — if the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in static global memory.
  - **local lifetime, static shape** — if the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine’s stack frame at run time.
  - **local lifetime, shape bound at elaboration time**

Arrays

• dope vector
  - collection of information about an array
    - dimensions
    - bounds (lower and upper)
  - record
    - offset to each field
    - good for runtime checks
    - can enhance efficiency by avoiding computations
    - origin: “having the dope on (something)”

Arrays

• stack allocation
  - conformant arrays
    - parameters to subroutines
    - dimensions not known
    - dope vector passed to help
Arrays

- Arrays are stored in contiguous memory locations.
- For multidimensional arrays, an ordering of dimensions must be specified:
  - Column-major - only in Fortran
  - Row-major
    - Used by everybody else
    - Important for element access

- To access elements in a single row:
  - Column-major
    - Cache may not be big enough to store next row item
  - Row-major
    - Effective
    - Reversed for column support, but row support seems to be used more often

- Two layout strategies for arrays:
  - Contiguous elements
  - Row pointers
- Row pointers
  - An option in C
  - Allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - Avoids multiplication
  - Nice for matrices whose rows are of different lengths
    - E.g. an array of strings
  - Requires extra space for the pointers

Arrays

- Heap allocation
  - Dynamic arrays
  - Dimensions not known until runtime
  - Dope vector can be maintained dynamically
Arrays

• address calculation
  – can compute some portions before execution
  – declaration
    \[
    A : \text{array}[I_1 ... I_d] \text{of array}[J_1 ... J_c] \text{of element type};
    \]
  – constants
    \[
    S_i = \text{size of element type}
    \]
    \[
    S = (I_1 - 1) \times J_1 \times S_i
    \]
  – address
    \[
    \text{address of } A
    = (I_2 - 1) \times S_i
    + (I_1 - 1) \times S
    + \text{address of } \text{array}[I_1 ... I_d] \text{of array}[J_1 ... J_c] \text{of element type};
    \]

Strings

• strings are really just arrays of characters
• they are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  – easier to provide these operations for strings than for arrays in general because strings are one-dimensional and (more importantly) non-circular

Sets

• possible implementations
  – bitsets are what usually get built into programming languages
  – intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
  – some languages place limits on the sizes of sets to make it easier for the implementer
    • for 32-bit int’s, bitset size is 500MB
    • for 64-bit int’s, bitset size enormous
Pointers and Recursive Types

• pointers serve two purposes
  – efficient (and sometimes intuitive) access to elaborated objects (as in C)
  – dynamic creation of linked data structures (recursive types), in conjunction with a heap storage manager

• several languages (e.g., Pascal) restrict pointers to accessing objects in the heap

Pointers and Recursive Types

• operations on pointers
  – allocation and deallocation of objects in the heap
  – dereferencing of pointers to access the objects to which they point
  – assignment from one pointer to another

• operations behave differently depending on whether the language employs a reference model or value model for names
  – functional languages use reference model

Pointers and Recursive Types

• reference model
  – ML example
    datatype chr_tree = empty | node of char * chr_tree * chr_tree;
  – example tree
    (#"X", node (#"Y", empty, empty), node (#"Z", empty, empty), node (#"W", empty, empty)))
  – block storage from heap

Pointers and Recursive Types

• reference model
  – Lisp example
    '(#\( #\( #\) #\) #\( #\) #\) #\( #\) #\( #\) #\))
  – each level of parentheses brackets the elements of a list
    • head and remainder of list
    • each node is denoted as a construct cell or atom
Pointers and Recursive Types

• references in functional languages are acyclic
  – circular references typically found only in imperative languages
• often need mutually recursive types
  – example: symbol table and syntax tree nodes need to refer to each other
  – if declared one at a time, not able to refer to each other

Pointers and Recursive Types

• tree types in value model languages
  – Pascal
    type chr_tree_ptr = 'chr_tree;
    chr_tree = record
      left, right : chr_tree_ptr;
      val : char
    end;
  – Ada
    type chr_tree;
    type chr_tree_ptr is access chr_tree;
    type chr_tree is record
      left, right : chr_tree_ptr;
      val : character;
    end record;

Pointers and Recursive Types

• mutually recursive types
  – ML example: symbol table and syntax tree nodes declared together
    datatype sym_tab_rec = variable of ...
    | type of ...
    | ...
    | subroutine of {code : sym_tree_node, ...}
    and sym_tree_node = expression of ...
    | loop of ...
    | ...
    | subr_call of (subr : sym_tab_rec, ...);

Pointers and Recursive Types

• tree types in value model languages (cont.)
  – C
    struct chr_tree {
      struct chr_tree *left, *right;
      char val;
    };
• for mutually recursive types
  – Pascal: forward references
  – Ada and C: incomplete data types

Pointers and Recursive Types

• allocation of space from the heap
  – Pascal
    new(chr.tree);
  – Ada
    my_ptr := new chr.tree;
  – C
    my_ptr = (struct chr_tree *) malloc(sizeof(struct chr_tree));
    * returns void*
  – C++, Java, C#
    my_ptr = new chr_tree( arg list );
Pointers and Recursive Types

- accessing objects pointed to by references
  - Pascal
    
    ```pascal
    my_ptr^.val := 'x';
    ```
  - C
    
    ```c
    (my_ptr).val = 'x';
    ```
  - Ada
    
    ```ada
    my_ptr->val = 'x';
    ```
  - or for structs
    
    ```ada
    my_ptr->val = 'x';
    ```
  - depends on context

Pointers and Recursive Types

- C pointers and arrays
  
  ```c
  int m;
  int *a;  /* pointer to integer */
  int b[10];  /* array of 10 integers */
  ```
  
  - all of the following are valid
    
    ```c
    1. a = b;  /* make a point to the initial element of b */
    2. n = a[3];
    3. n = *a[3];  /* equivalent to previous line */
    4. n = b[3];
    5. n = *(b+3);  /* equivalent to previous line */
    ```

Pointers and Recursive Types

- C pointers and arrays closely linked
  
  ```c
  int *a == int a[]
  int **a == int *a[]
  ```

- some subtle differences
  
  - a declaration allocates an array if it specifies a size in the first dimension
  
  ```c
  int **a, int *a[];  // pointer to pointer to int
  ```
  
  - otherwise it allocates a pointer
  
  ```c
  int *a[n];  // n-element array of row pointers
  int a[n][m];  // 2-d array
  ```

Pointers and Recursive Types

- most languages blur the distinction between l-values and r-values by implicitly dereferencing variables on right-hand side of assignment

Pointers and Recursive Types

- subscript operator [ ]

  ```c
  E1[E2]
  ```

  - equivalent to
  
    ```c
    (*((E1)+(E2)));
    ```

  - order does not matter
  
    ```c
    A[3]
    3[A]
    ```

Pointers and Recursive Types

- declaring an array requires all dimensions except the last
  
  - invalid
  
    ```c
    int a[];
    int (*a)[];
    ```

  - C declaration rule: read right as far as you can (subject to parentheses), then left, then out a level and repeat
  
    ```c
    int *a[n];  // n-element array of pointers to integer
    ```

    ```c
    int (*a)[n];  // pointer to n-element array of integers
    ```
Pointers and Recursive Types

• when a heap-allocated object is no longer live, space must be reclaimed
  – stack objects reclaimed automatically
  – heap space can be reclaimed automatically with garbage collection

Pointers and Recursive Types

• some languages require heap space to be reclaimed by the programmer
  – Pascal
    
    dispose(my_ptr);
  – C
    
    free(my_ptr);
  – C++
    
    delete my_ptr;
  – if not reclaimed, results in memory leaks

Pointers and Recursive Types

• dangling reference
  – live pointer that no longer points to a valid object
  – only in languages that have explicit deallocation

Pointers and Recursive Types

• dangling reference examples
  – subroutine returns pointer to local variable
  – active pointer to space that has been reclaimed
    • even if the pointer were changed to NULL, other pointers might still refer to this space
    • changes may read or write bits that are parts of other objects

Pointers and Recursive Types

• explicit reclaiming can be a burden on programmers and a source of memory leaks
• alternative: garbage collection
  – automatic reclaiming of heap space
  – required for functional languages
  – found in Java, Modula-3, C#
  – difficult to implement
  – convenient for programmers: no dangling pointers
  – can be slow

Pointers and Recursive Types

• several types of garbage collection
  – reference counts
  – mark-and-sweep
  – pointer reversal
  – stop-and-copy
  – others
Pointers and Recursive Types

- reference counts
  - one way to designate an object as garbage is when no pointers to it exist
  - place a reference count in each object that keeps track of the number of pointers to it
  - when object is created, reference count is set to 1
  - when pointers are assigned or re-assigned, count is incremented/decremented
  - when count reaches 0, space can be reclaimed

Pointers and Recursive Types

- the heap is a chain of nodes (the free_list)
- each node has a reference count (RC)
- for an assignment, like $q = p$, garbage can occur

Pointers and Recursive Types

- disadvantages
  - failure to detect inaccessible circular chains
  - storage overhead created by appending an integer reference count to every node in the heap
  - performance overhead whenever a pointer is assigned or a heap block is allocated or deallocated

Pointers and Recursive Types

- reference counts (cont.)
  - reference count also decremented upon subroutine return if local pointers involved
  - system must run recursively, so that any pointers in reclaimed space are accounted for
  - pointers must otherwise be initialized to null
  - works well with strings, which have no circular dependencies

Pointers and Recursive Types

- some garbage collection schemes use tracing
  - instead of counting the number or references to an object
  - find all good space by following valid pointers that are active
  - work recursively through the heap, starting from external pointers
**Pointers and Recursive Types**

- **mark-and-sweep**
  - 3 main steps executed when space is low
    1. all blocks in the heap marked with 0
    2. follows pointers outside the heap and recursively explores all heap space; each newly discovered block marked with 1
    3. final walk through heap to remove every block still marked with 0

- **now free_list is restored and**
- **the assignment** `q=new node()` **can proceed**

- **pointer reversal**
  - to avoid large stack during recursive search, use pointers already in heap space to work back up the chain
  - such pointers oriented in the wrong direction
  - can be reversed to point backward in the “stack”
  - collector keeps track of current block and the block from whence it came

- **advantages over reference counting**
  - reclaims all garbage in the heap
  - only invoked when the heap is full

- **issues with mark-and-sweep**
  - system must know beginning and ending points of blocks
    - each block must contain a size
  - collector must be able to find pointers contained in each block
  - recursive nature requires stack space
  - can be time-intensive

- **heap exploration using pointer reversal**
Pointers and Recursive Types

- stop-and-copy (aka copy collection)
  - heap space divided into two halves
    - only one is active at any given time
    - when one half nearly full, collector recursively explores heap, copying all valid blocks to other half
    - defragments while copying
    - pointers to blocks changed during copy
    - when finished, only useful blocks in heap; other half is no longer used
    - on next collection, halves are reversed

Pointers and Recursive Types

- accessible nodes copied to other half
  - the accessible nodes are packed, orphans are returned to the free_list, and the two halves reverse roles

Pointers and Recursive Types

- other techniques
  - generational collection
    - since most dynamically allocated blocks are short-lived, newer ones stored in half the heap
    - this half is checked first if garbage collection necessary
    - during garbage collection, valid blocks transferred to long-term heap
    - system must examine long-term heap if more space needed, and to reduce memory leaks
  - conservative collection
    - if blocks in the heap do not state explicitly the location of pointers, any space that “looks like” a pointer can be considered such and used with mark-and-sweep
    - may leave some useless blocks, but guaranteed not to delete currently used blocks

Pointers and Recursive Types

- triggered by q=new node() and free_list outside the active half:

Pointers and Recursive Types

- issues with stop-and-copy
  - only half of the heap can be used at any one time
  - incurs additional overhead
  - can be time-intensive

Pointers and Recursive Types

- summary
  - modern algorithms are more elaborate
  - most are hybrids/refinements of the above three
  - in Java, garbage collection is built-in
    - runs as a low-priority thread
    - also, System.gc may be called by the program
  - functional languages have garbage collection built-in
  - C/C++ default garbage collection to the programmer
Lists

- A list is defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - Ideally suited to programming in functional and logic languages, which rely on recursion
    - Lisp: a program is a list, and can extend itself at run time by constructing a list and executing it
    - Lists can also be used in imperative programs

- Since everything in Lisp is a list, differentiation required between lists to be evaluated and constant lists

Lisp list constructors

```lisp
(list 'a 'b 'c)    => (a b c)
(cdr '(a b))      => a
(car nil)         => nil
(cdr '(a b c))    => (b c)
(car '(a))        => a
(car nil)         => nil
(append '(a b) '(c d)) => (a b c d)
```

Lists

- List operations tend to generate garbage
  - Useful to have automatic garbage collection
- Some languages (e.g., ML) require lists to contain items of the same type
- Other languages (e.g., Python, Lisp) allow lists to contain items of differing types
  - Implemented as chain of two pointers: one to element and one to next block

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - Interactive I/O generally implies communication with human users or physical devices
  - Files generally refer to off-line storage implemented by the operating system
- Files may be further categorized into
  - Temporary
  - Persistent

Equality Testing and Assignment

- When an expression is encountered testing for equality, how should it be interpreted?
  - For primitive types, easily defined
  - Example: s = t for character strings
    - Are s and t aliases?
    - Occupy the same storage that is bit-wise identical over its full length?
    - Contain the same sequence of characters?
    - Would appear the same if printed?
  - Shallow vs. deep comparison
• Scheme has distinct equality-testing functions for each

- behavior
  
  (eq? a b) \Rightarrow \#t  \quad \text{true if a and b refer to the same object.}
  
  (eqv? a b) \Rightarrow \#f  \quad \text{false if a and b are not equal, possibly due to different representations.}
  
  (equal? a b) \Rightarrow \#f  \quad \text{false if a and b have the same recursive structure.}

• deep assignments rare in any language