Overview of Imperative Paradigm

- Names, location, reference — page 3
- Expressions — page ??
- Control — page ??
- Types, polymorphism — page ??
- Block structure, parameter passing, closures — page ??
- Exception handling — page ??
- Abstract data types and modules — page ??
- Computer algebra systems?
- Scripting languages?
- Object-oriented languages — page ??
Overview of Names, Locations, Pointers

- Names and what they refer to
- l-value versus r-value, l-valued functions
- Declarations, scope, extent, storage classes
- Pointers, dangling references, garbage collection
- Expressions, referential transparency
Names

Naming the things we compute with is a psychological necessity. But names are more subtle than first meets the eye. Also, from the implementation point of view, computers don’t really “like” names.
Sebesta, 11th, 5.2.2 Name Forms, page 199 ff.
Identifier Syntax

Ada: letters, digits, underscores; arbitrary length, must begin with letter, only one underscore between letter or digit, case ignored.
Java: letters (Unicode), digits, underscores and dollar signs; arbitrary length, should begin with a letter.
SML: letters, digits, symbols, apostrophe; arbitrary length, distinguishes different categories of identifiers: value, type, equality types, etc.
Identifiers

Fortran 90: letters, digits, underscore; up to 31 characters in length, must begin with a letter
C: letters, digits, underscores; must not begin with a digit. Avoid beginning with underscores or bad things can happen. K&R state the first 8 characters are significant. C99 requires 63 characters be significant and case sensitive.
C++: letters, digits, underscores; must not begin with a digit.
C#: allows keywords as identifiers (because of interoperability with other languages) if they are prefixed with @.
In lambda when ignoring the argument is often the practice to use that pattern ‘_’ when the argument is unused in the body. In Java is is tempting to use the identifier ‘_’.

“I don’t want to give this argument a name” (such as catch parameters that is never used)

Java remove the underscore as a legal single character identifier for that it could be used in

After Java 9 “We are now free to complete the planned rehabilitation of underscore to indicate an unused lambda, method, or catch formal parameter.” From JEP 302: Lambda Leftovers
Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | unicode_digit } .
```

Here are some examples:

- a
- _x9
- ThisVariableIsExported
- αβ

Some identifiers are predeclared.
Identifiers – Go

const abc_1 = 1 // Legal identifier
const _ = 2 // Legal, but "cannot use _ as value"
const $ = 3 // '$' not legal in identifier
const for = 4 // ERROR: keywords are reserved
var nil = 5 // predeclared literal
var true = 5 // predeclared literal
var false = 6 // predeclared literal
const int64 = 7 // predeclared type
const make = 8 // predeclared function
const abc_9 = 9
Blank identifier
The blank identifier is represented by the underscore character __. It serves as an anonymous placeholder instead of a regular (non-blank) identifier and has special meaning in declarations, as an operand, and in assignments.
Most languages say the two identifiers are the same if they are made of the same character code points.

But some code points are just variants—think upper case and lower case variants of the same Latin, Greek, or Cyrillic letters.

This conflicts with the sensible style requirement that identifiers should be psychologically different for readability. Might might easily confuse abCde with abCde or kLmn with kLmn.

A few languages like Ada ignore case in identifiers.
Identifiers – Unicode

Unicode Report #31

Problems: language direction makers, combining forms, fullwidth, halfwidth, variants
A sigil (/ˈsɪdʒəl/ or /ˈsɪgəl/; plural sigilia or sigils) is a symbol attached to a variable name, showing the variable’s datatype or scope.

Wikipedia: Sigil, from the Latin meaning a "little sign", means a sign or image supposedly having magical power. In 1999 Philip Gwyn adopted the term "to mean the funny character at the front of a Perl variable".
In my opinion, distinguishing types (a la Hungarian notation) is best done by localizing scope and selecting names that suggest their very specific purpose in the program. The purpose also helps with a variable’s extent. And scope should be made clear by the language. IDEs that provide easy access to an identifiers definition (and to its associated comments) are a great help. Fortran use of implicit declarations (I–N are variables that have integer type) seems a particularly bad.
Identifiers – Sigil

Some languages use the technique more successfully. In Ruby "$" is prefixed to global variables, "" (one at) is prefixed to instance variables, and "" is prefixed to class variables. By convention, the suffix "?" indicates a method returning a boolean value; and "!" indicates that the method has a side effect and so is not safe.

In many dialects of BASIC, string variables are suffixed with "$"

OCaml; capital letter ; module or module type or constructor

OCaml and SML; apostrophe ; type variable

SML; double apostrophe ; type variable allowing equality
An identifier is a name used in a program. A variable is an identifier for a location. Do not confuse the two terms. Tip: Do not use the word “variable.”
Syntax of Assignment Statement

Imperative paradigm, dominate features: :=, goto, if

Examples of the syntax of assignment:

- $A := 3$ Pascal, Ada, Modula-3, ALGOL 68
- $A = 3$ Fortran, PL/I, SNOBOL4, C, Java
- $A <- 3$ Smalltalk, Mesa, APL
- $A =. 3$ J
- $3 -> A$ BETA
- MOVE 3 TO A COBOL
- (SETQ A 3) LISP
Alan Jan Perlis (1922–1990)

An Englishman by birth he worked at Princeton and Purdue. First editor-in-chief of *Communications of the ACM*. First Turing award recipient. Perlis has been an influence in the area of programming techniques and compiler construction.
Founded the Digital Computer Center at Purdue University in 1952 and developed IT. In 1956 founded the Computation Center at Carnegie Institute of Technology (now Carnegie-Mellon University). He was involved in defining ALGOL 58 and ALGOL 60. First editor of *Communications of the ACM*, first President (1962-1964) of the ACM, first (1966) recipient of the ACM’s Turing Award.
“Epigrams on Programming" is an article by Alan Perlis published in SIGPLAN Notices 17, September 1982. The epigrams are a series of short, humorous statements about computers and programming, which are widely quoted.

41. Some programming languages manage to absorb change, but withstand progress.
48. The best book on programming for the layman is “Alice in Wonderland”; but that’s because it’s the best book on anything for the layman.
117. It goes against the grain of modern education to teach children to program. What fun is there in making plans, acquiring discipline in organizing thoughts, devoting attention to detail and learning to be self-critical?
“You are sad,” the Knight said in an anxious tone: “let me sing you a song to comfort you. 
... The name of the song is called “‘Haddocks’ Eyes’.”
“Oh, that’s the name of the song, is it?” Alice said, trying to feel interested.
“No, you don’t understand,” the Knight said, looking a little vexed. “That’s what the name is
called. The name really is ‘The Aged Aged Man’.”
“Then I ought to have said ‘That’s what the song is called’?” Alice corrected herself.
“No, you oughtn’t: that’s quite another thing! The song is called ‘Ways And Means’: but
that’s only what it’s called, you know!”
“Well, what is the song, then?” said Alice, who was by this time completely bewildered.
“I was coming to that,” the Knight said. “The song really is ‘A-sitting On A Gate’: and the
tune’s my own invention.”
The white Knight’s song by Lewis Carroll from *Through the Looking Glass*. Lewis Carroll is a pseudonym for Charles Lutwidge Dodgson. Carroll was a logician and uses language in precise way; this can amuse people who are generally happier with imprecision.
Use/mention distinction.

Carroll distinguishes between naming and calling. Unusual but not unheard of. “Her name is “Veronia,” but she is called Franky.”

The Piano Sonata No. 14 in C-sharp minor “Quasi una fantasia”, Op. 27, No. 2, popularly known as the Moonlight Sonata, is a piano sonata by Ludwig van Beethoven.
L-value

Names in computer programs written in conventional languages are schizophrenic. They refer to two things at different times: a location and a value. The $X$ on the left-hand side of the assignment $X := X + 1$ refers to the location of $X$; the $X$ on the right-hand side refers to the value of $X$. We call the location referent of an identifier the \textit{l-value} and the value referent the \textit{r-value}, because we are so accustomed to having identifiers stand for their value on the right-hand side and for their location on the left-hand side.
“My great fear is that we’ve in fact been visited by intelligent aliens,” De-Grasse Tyson said to MSNBC host Chris Hayes in 2014. “But they chose not to make contact, on the conclusion that there’s no sign of intelligent life on Earth.”

\[ x = x + 1 \] C programming language
A few languages distinguish between the l-value and the r-value of a variable.

\[
\begin{align*}
X & := \ .X + 1 & \text{BLISS} \\
X & := \ !X + 1 & \text{ML}
\end{align*}
\]
Expressions standing for an r-value are common. Some typical r-valued expressions:

x
x+2
3*x+2
A[2*i]**2
r.f / 3.4
i=3? a[j] : x

More about expressions later . . .
L-valued Expressions

L-valued expressions are usually more limited, but there are some:

x
A[2*i]
A[i][j]
r.f
r.A[i+4]
A[i][j].f
if i=3 then a[j] else x
L-valued Functions

Functions can return l-values in some languages.

// l-value as result of function call in C++
int a[10];        // declare an array
int& f(int i) {return (a[i]);}   // define f
f(5) = 17;        // a[5] := 17

(* l-value as result of function call in ML *)
val x = ref 1;    (* declare a variable *)
val y = ref 2;    (* declare a variable *)
fun f (n) = if (n mod 2)=0 then x else y;
f(3) := 4;        (* y := 4 *)
The prefix and postfix ++ and -- operators are require an l-valued operand. They change the value stored in the l-value and then return the value there. They either return the value before the operation or after.
Thompson went a step further by inventing the ++ and -- operators, which increment or decrement; their prefix or postfix position determines whether the alteration occurs before or after noting the value of the operand. They were not in the earliest versions of B, but appeared along the way. People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular. This is historically impossible, since there was no PDP-11 when B was developed. The PDP-7, however, did have a few ‘auto-increment’ memory cells, with the property that an indirect memory reference through them incremented the cell. This feature probably suggested such operators to Thompson; the generalization to make them both prefix and postfix was his own. Indeed, the auto-increment cells were not used directly in implementation of the operators, and a stronger motivation for the innovation was probably his observation that the translation of ++x was smaller than that of x=x+1.

The Development of the C Language, Dennis M. Ritchie, 1993
A binding is an association of a name (an identifier) to something. An environment is collection of bindings.

Binding times:

- Language definition
- Implementation
- compile time
- Link time
- run time
  - elaboration time (at some indeterminate point before the execution of affected statements a declaration may require preparation, e.g., a location initialized and object created, etc.);
  - (dynamic) load time; Java’s execution of static initialization code for each loaded class
  - statement execution time
There is always a trade-off between efficiency (early) versus flexibility (late). Generally speaking, things that happen at compile time are described as *static* and those a run time (during the execution of the program) are *dynamic*.

It is extremely important to distinguish what happens at compile time for those tings that happen at runtime.
Efficient versus Flexible

F: constant INTEGER := 4;
.. F*X .. -- * can be replaced by left shift 2

C: constant INTEGER := Get_Input_From_User;
In many languages a named constant is required to have a value that can be
determined at compile time. Usually the expression that specifies the constant’s
value is permitted to include only other known constants and built-in functions
and arithmetic operators. Named constants of this sort, together with constant
literals, are sometimes called manifest constants or compile-time constants.
Manifest constants can always be allocated statically, even if they are local to
a recursive subroutine: multiple instances can share the same locations.

A *construct* of a language is a syntactically meaningful subpart of a language formed (constructed) in accordance with the syntactic rules of a language and has a significant and coherent purpose. Typical examples in programming languages include types, statements, expression, and control constructs.
Rough anatomy of a programming language:

**Statement.** A **statement** is a construct in a programming language that performs some action or governs the control flow.

**Expression.** An **expression** is a construct in a programming language the stands for or denotes some value or object of computation.

**Declaration.** A **declaration** is a construct in a programming language which announces or introduces a name (an identifier) for use in the program.

*Type.* Comes later.

*Patterns* — a special part of declarations in some languages — we’ll encounter in functional languages.)
Declaration. A **declaration** is a construct in a programming language which announces or introduces a name (an identifier) for use in the program. It may also associate some property with the name. There are declarations for values (numbers, strings, etc.), locations (variables), types, functions, subprocedures, exceptions, modules, etc.

Pi : constant Float := 3.14159;
Prompt : constant String := "=>";
X : Integer;
type Color = (Red, Green, Blue);
procedure P is begin null end P;
E : exception

Maxim: You must declare identifiers before you can use them.
The Word “declaration”

A dictionary definition:

*the formal announcement of the beginning of a state or condition. For example, “the declaration of war”*

Synonyms: proclamation, notification, announcement, revelation, disclosure
Do you have anything to declare?
¿Tienes algo que declarar?
Haben Sie etwas anzugeben?
У вас есть что-либо, подлежащее декларированию?
(U vas yest' chto-libo, podlezhashcheye deklarirovaniyu?)
هل لديك أي شيء تعلنه؟
(Hal ladayk 'ayu shay' latuelinah?)
你有需要申报的东西吗？
(Nǐ yǒu xūyào shēnbào de dōngxī ma?)
क्या आपके पास घोषित करने के लिए कुछ है?
(Kya aapake paas ghoshit karane ke lie kuchh hai?)
Recursive Declarations

Ada:

```ada
type Node;
type Tree is access Node;
type Node is record
    Item : Integer; Left, Right : Tree;
end record;
```

Modula-3:

```modula-3
TYPE Tree = RECORD
    item: INTEGER; left, right: REF Tree
END;
```
Recursive Programs

recurse/nested.adb
Pascal forward

```pascal
type
  company = record CEO: ^person; end;
  person  = record employer: ^company; end;

procedure Q (A, B: integer); forward;
procedure P (C: integer);
begin
  Q (3,4);
end;
procedure Q; (* args not repeated *)
begin
  P (17);
end;
```
A namespace is an abstract container or environment created to hold a logical grouping of unique identifiers or symbols (i.e., names). The same identifier can be independently defined in multiple namespaces. That is, the meaning associated with an identifier defined in one namespace may or may not have the same meaning as the same identifier defined in another namespace.
Namespaces

```java
package Reuse;

class Reuse {
    Reuse Reuse;
    Reuse Reuse (Reuse Reuse) {
        Reuse: for (;;) {
            if (Reuse.Reuse(Reuse)==Reuse.Reuse)
                break Reuse;
        }
        return Reuse;
    }
}
```

There are six different namespaces in Java: packages, types (classes), field names, method names, local variables / formal parameters, and labels.
Overloading

*Overloading* happens when one name with more than one meaning which can be disambiguated by context.

Names other than the names of subprocedures can be overloaded.

More about overloading later under the topic of polymorphism.

```
declare
   type Month is (Jan, Feb, Oct, Nov, Dec);
   type Base is (Dec, Bin, Oct, Hex);
   X: Boolean;
   M: Month;
begin
   M := Dec;  -- *Exactly one possibility*
   X := Oct < Dec;  -- *Ambiguous*
end;
```

Scott, 3rd, page 146.
Scope and extent

Scope. The *scope* of a declaration is the portion of the program text in which the identifier introduced in the declaration is visible; i.e., has the attributes given in by the declaration. We also speak of *visibility*.

More about scope after introducing “block structure” . . .

Extent. The *extent* of a location in memory is the period of program execution in which the location can be accessed by some name or expression in the program.
Extent categories: static, local, dynamic.
Storage Class

- static (C), own (ALGOL 68);
  global, allocated once
  ```c
  void f () {
    static int count = 0;
    count++;
  }
  ```

- local, automatic (C, PL/I), loc (ALGOL 68);
  during lifetime of procedure invocation, very structured, very efficient

- dynamic, based (PL/I), heap (ALGOL 68).
  explicitly controlled by programmer, new and delete
Static Allocation

Local variables created when their subroutine called, destroyed on exit. What about recursive subroutines? Not possible, because no place to store local variables pertaining to different activations of the same procedure. Scott, 2nd, Figure 3.1, page 110?
Static allocation of local variables not an option. LIFO order of procedure calls suggests a stack.
Stack Allocation

Activation record. Register save, temporaries, bookkeeping, local variables. Scott, 2nd, Figure 3.2, page 110?
Runtime Organization

The stack
(local variables)

The heap
(dynamically allocated variables)

Program code
(read-only)

Global variables

Runtime organization of a program in memory
Runtime system. The runtime system of a program is the collection of facilities supporting the execution of the program. The language translation system (compiler and linker) adds the code for these facilities along with the code produced by translating the user’s program to produce the executable module. The runtime system may perform many different tasks: checking narrowing, checking subtype constraints, searching for exception handlers, allocating storage from the heap, controlling interaction between tasks, performing garbage collection.
Meaning of Identifiers

- names
- environment
- locations
- values
- state

Diagram showing the relationship between names, environment, locations, values, and state.
Storage Models

Three types:

- let semantics – used for constants
- pointer semantics – copy pointers
- storage semantics

Parts of the model:

- environment $\rho$ – mapping of names to values and locations
- store $\sigma$ – mapping of locations to values
Let semantics

\[
\text{let } A = B \\
\rho_f(A) \leftarrow \rho_0(B)
\]

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\rho_0(A) = 4)</td>
<td>(\rho_f(A) = 7)</td>
</tr>
<tr>
<td>(\rho_0(B) = 7)</td>
<td>(\rho_f(B) = 7)</td>
</tr>
</tbody>
</table>
**Pointer semantics**

\[
A : \leftarrow B
\]

\[
\rho_f(A) \leftarrow \rho_0(B)
\]

NB $\sigma$ unchanged.

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_0(A) = \alpha_1$</td>
<td>$\rho_f(A) = \alpha_2$</td>
</tr>
<tr>
<td>$\rho_0(B) = \alpha_2$</td>
<td>$\rho_f(B) = \alpha_2$</td>
</tr>
<tr>
<td>$\sigma_0(\alpha_1) = 4$</td>
<td>$\sigma_f(\alpha_1) = 4$</td>
</tr>
<tr>
<td>$\sigma_0(\alpha_2) = 7$</td>
<td>$\sigma_f(\alpha_2) = 7$</td>
</tr>
</tbody>
</table>
Storage semantics

\[ A := B \]

\[ \sigma_f(\rho_0(A)) \leftarrow \sigma_0(\rho_0(B)) \]

NB \( \rho \) unchanged.

<table>
<thead>
<tr>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho_0(A) = \alpha_1 )</td>
<td>( \rho_f(A) = \alpha_1 )</td>
</tr>
<tr>
<td>( \rho_0(B) = \alpha_2 )</td>
<td>( \rho_f(B) = \alpha_2 )</td>
</tr>
<tr>
<td>( \sigma_0(\alpha_1) = 4 )</td>
<td>( \sigma_f(\alpha_1) = 7 )</td>
</tr>
<tr>
<td>( \sigma_0(\alpha_2) = 7 )</td>
<td>( \sigma_f(\alpha_2) = 7 )</td>
</tr>
</tbody>
</table>
Assignment

```c
int A;
int *C=&A, *D=new int;  /* pointers in C++ */
*C = 17;
C = D;
*D = 97;    /* changes C! */

int[] A = new int [5];
int[] B = new int [17];
A = B;
B[8] = 97;    /* changes A! */

declare
A,B: array (1..5) of Integer;
begin
    A := B;  -- copies the array
end;
```
Assignment in C++

```cpp
void f () {
    int i=1, j=2; // Declare 2 integer variables
    int& r = i;   // Declare int ref & initialize
    r = j;       // Assign contents of j to r
}
```

r=i: r denotes same location as i (pointer semantics)  
r=j: r contains same value as j (storage semantics)
Model of Example C++ Program

<table>
<thead>
<tr>
<th></th>
<th>int i, j</th>
<th>i=1; j=2</th>
<th>int&amp; r=i</th>
<th>r=j</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho(i) )</td>
<td>( \alpha_1 )</td>
<td>( \alpha_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \rho(j) )</td>
<td>( \alpha_2 )</td>
<td>( \alpha_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \rho(r) )</td>
<td>( \alpha_1 )</td>
<td>( \alpha_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \sigma(\alpha_1) )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( \sigma(\alpha_2) )</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Pointer semantics leads to confusion, because (like the white knight) we have several names for the same (or seemingly the same) thing.
Changing one thing changes another confusion.
There is a good solution: immutability.
But there is a cost: copying.
Integers are an example of an immutable value. An *immutable* object is one that no operation can change. Adding one to an integer does not change it; the result is simply another integer. On the other hand, we typically view the updating of an element of an array $A$ or a field of a record as changing $A$, not creating a new array. Thus an array is a *mutable* object, an object that “has a state which may be modified by certain operations without changing the identity of the object.”

Do not confuse immutable with the term *constant*. An identifier is said to be *constant* if it always refers to the same object (immutable or not). An object is said to be *immutable* if no operations can change it.
A const method is one that does not alter the object.

class Car {
public:
    void reset () { speed=0; }
    void delta (int x) { speed+=x; }
    int getSpeed () const { return speed; }
    bool isForward () const { return speed>0; }
private:
    int speed;
};
class Car {
    private int speed = 0;
    public void reset () { speed=0; }
    public void delta (int x) { speed+=x; }
    public int getSpeed() { return speed; }
    public boolean isForward(){return speed>0;}
}

Primitive types are immutable and assignment is the usual storage semantics. Everything else is (potentially) mutable and implemented using pointers, so pointer semantics.

What if you want storage semantics for (non-primitive) objects in Java? What if you want to copy the value and not the reference?

Copying is supported in Java through the `clone` method of the class `Object` and the peculiar interface `Cloneable`. (It is a tagging or marking interface since it requires the implementation of no methods.)
Java

class C implements Cloneable {
    int x = 4;                 // even x gets copied
    public Object clone () throws Exception {
        return super.clone();  // field-by-field copy
    }
}

C c = new C ();
C d = c;                // pointer semantics
System.out.println (d.x);
c.x = 3;                 // "d" is changed
System.out.println (d.x);
d = (C) c.clone ();      // make a copy
c.x = 2;
System.out.println (d.x);  // now "d" unchanged
Notice how pointer semantics violates the axiom of assignment: a change to c does not leave d alone. Even worse, assigning to c does not even change c (it is still the same object).

Notice that we did not create garbage by discarding the first contents of d. c still points to that object, so no garbage was created.
Pointers

Dynamic allocation. The allocation of computer storage for program variables at run-time, possibly because the number of required memory locations was not known at compile-time.

Heap. The section of memory used by the executing program available for dynamic allocation of storage for program variables.

Storage management. The organization and maintenance of computer memory for program variables by the executing program.

Alias. Two references to the same location.

Garbage. Locations allocated by the runtime system, but now no longer accessible by the program.
Primitive and Reference Variables

Picture from cse1002.
Pointers

When a storage location is not just denotable (by l-expressions), but can be a value (r-value) itself, then we have a pointer. Pointers may be declared statically, but the storage locations may be created and disposed dynamically under the control of the program.

Why pointers? Dynamically changing data, recursive data types. Not necessarily a good idea in high-level languages. Hoare said it was a “seriously retrograde step.”

```c
struct tree {
    int n; struct tree *left; struct tree *right;
};

class Tree {
    int n; Tree left, right;
}

datatype tree = empty | node of int * tree * tree
```
“Their introduction into high-level languages has been a step backward from which we never recover.”
“there are many reasons to believe that the introduction of references into a high-level language is a seriously retrograde step”
(Hoare biography earlier with axiomatic semantics.)
Recursive data structures can be quite complex (e.g., finger trees). Recursive data structures (by definition) have values that can be expressed without using pointers.

```ocaml
datatype tree = empty | node of int * tree * tree
val t = node (3, node (2, empty), empty)
```

In memory a tree looks like a jumble of pointers, but there are no cycles. Graphs have cycles and they are not definable recursively. Hence, no obvious language of canonical expressions for representing them. Graphs, like trees, look like a jumble of pointers in memory, but they have potential cycles. Hence, no obvious way to construct a graph value except using pointers.
(But see neato in GraphVis for a language for describing graph layout.)
Does Java have Pointers?

There is not a data type for pointers in Java because all non-primitive data values are references. No operations on the pointer data type (as in C/C++). However, explicit allocation is still required. If you forget to allocate storage for “pointer” variables, then you get a null pointer error (a common problem in Java). But deallocation is unnecessary as Java does garbage collection.

Example: java/programs/class/Tree.java

Assignment for objects is pointer semantics (as we have seen). Other alias problems (parameter passing) make it important to understand pointers in order to program in Java.
Pointers

It is possible to create inaccessible locations with pointers: `new(p); p:=nil`. Whatever storage location was obtained for the program is now unusable and said to be garbage. Inaccessible locations can be detected by the run-time system and returned to a pool of free storage locations. This is called _garbage collection_. Worse, it might be possible to still refer to inaccessible locations even after the OS has reused the locations. References to inaccessible locations are said to be _dangling references_.

```pascal
var p,q: ^integer;
begin
    new(p);
    q:=p;
    dispose(p)
    (* q may point to garbage now *)
end;
```
C/C++ is notorious

Hanging on to the address of a local variable.
How does & make aliasing worse? Now you can alias local variables too!

/* & -- ‘address of’ operator */

int *f() { int i; return &i; }
int *i = f();
Algol 68 addresses the problem of dangling references to stack objects by forbidding a pointer from pointing to any object whose lifetime is briefer than that of the pointer itself. Unfortunately, this rule is difficult to enforce. Among other things, since both pointers and objects to which pointers might refer can be passed as arguments to subroutines, dynamic semantic checks are possible only if reference parameters are accompanied by a hidden indication of lifetime. Ada 95 has a more restrictive rule that is easier to enforce: it forbids a pointer from pointing to any object whose lifetime is briefer than that of the pointer’s type.

Scott, Section 7.7.2 Dangling Reference, page 392.
Extent

Ada examples:

ada/programs/access/illegal_1.adb
ada/programs/access/illegal_2.adb
ada/programs/access/outer.adb
access/outer.adb

We will see a similar problem later in the context of procedures. See the “Blocks” chapter.
Many computer languages require garbage collection, either as part of the language specification (e.g., Java, C#, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be garbage collected languages. Other languages were designed for use with manual memory management, but have garbage collected implementations available (e.g., C, C++). Some languages, like Ada, Modula-3, and C++/CLI allow both garbage collection and manual memory management to co-exist in the same application by using separate heaps for collected and manually managed objects.
Garbage Collection

Programmer reclamation of objects in the heap is a major source of bugs (memory leaks and dangling references). The code required is difficult to design, implement, and maintain.

**Garbage Collection.** Garbage collection is a task of a programming language’s runtime system in which dynamically allocated locations that are no longer accessible by the program are identified and reclaimed. Without garbage collection, a program might run out of unused memory locations for dynamically allocated data.

Programmers are *not* said to do garbage collection, only the runtime system does garbage collection. Sometimes programmers explicitly allocate and deallocate memory, but that is different as programmers are suppose to already know what memory may be accessed in the future.
Garbage Collection References

Webber, Section 14.5 Current Heap Links, pages 259–268.
Sebesta 7e, Section 6.9.9.3 Heap Management, pages 301–305.
Scott 4e, Section 8.5.3 Garbage Collection, pages 389–398.
Tucker & Noonan, Section 5.7 Memory Leaks and Garbage collection, pages, 143–150.
Wilson, “Uniprocessor Garbage Collection Techniques”
Garbage Collection

Three approaches:
1. Reference count
2. Mark, sweep
3. Stop-and-copy

Two important strategies
1. Generational garbage collection
2. Conservative garbage collection
This is an example of a genre of jokes told at the MIT AI Lab. The original koans were composed by Danny Hillis, who would later found Connection Machines, Inc. David Moon, of the now defunct Symbolics, Inc., wrote much of the Lisp Machine Manual.

One day a student came to Moon and said, “I understand how to make a better garbage collector. We must keep a reference count of the pointers to each of the cans.” Moon patiently told the student the following story:

“One day a student came to Moon and said, ‘I understand how to make a better garbage collector...’
Primitive and Reference Variables
Reference Count

Store a count of the references to the object along with the object. Increment the count when a new reference refers to the object. Decrement the count when a reference no longer refers to that object. When the count goes to zero, reclaim it, and decrement count for all descendants.
Requires extra space and is fooled by circular data structures.
Scott, Figure 7.19, page ?.
Reference Counting

On the whole, the problems with reference counting outweigh its advantages, and it is rarely used for automatic storage management in programming language environments.

Apple, 2nd, page 264.
Divide heap into two regions of equal size (the “to-space” and the “from-space”). When half is filled up, copy all the reachable data into the free half. Then swap halves. Compaction is very easy this way.
Generational Garbage Collection

The *generational hypothesis* states that most of the allocated objects die young. See: David Michael Ungar, 1984. DOI 10.1145/800020.808261
Generational Garbage Collection

Empirical studies have shown that the extent of most dynamically allocated memory is either very short or very long. Don’t bother collecting anything but recently allocated objects. Any data that survives more than one collection is assumed to be permanent. Heap divided into half: permanent and non-permanent, or even many generations $G_0$, $G_1$, $G_2$, . . . .

Each older generation should be exponentially bigger than the previous one. If $G_0$ is half a megabyte, then $G_1$ should be two megabytes, $G_2$ should be eight megabytes, and so on. An object should be promoted from $G_i$ to $G_{i+1}$ when it survives two or three collections of $G_i$. 
Generational
Conservative

Assumption: the runtime system expends a lot of bookkeeping effort to keep track of references to the heap (in order to know where to begin marking the accessible locations in the heap).
Premise: references to the heap can often be easily distinguished from other values (like integers). Often they must be big numbers divisible by 4.
Idea: Be conservative, just assume that any such values in the program are pointers and mark them (and their descendants) to keep. No harm (if no pointers are missed) and the effort of determining the real pointers is spared.
Memory Leak

An error in a program, system provided libraries, or runtime system’s storage management that causes it to fail to reclaim discarded memory, leading to the eventual collapse of the program due to running out of unused memory.
The following non-standard command-line flags are recognized by the VM.

- `XX:+UseConcMarkSweepGC` This flag turns on concurrent garbage collection. The collector executes mostly concurrently with the application. It trades the utilization of processing power that would otherwise be available to the application for shorter garbage collection pause times.

- `XX:+UseParallelGC` This flag enables garbage collection to occur on multiple threads for better performance on multiprocessor machines.
A similar sort of problem concerns initialization of variables. How is one to know if a variable has been initialized. Java requires the program variables to be initialized so obviously that data-flow analysis will verify that the variable has been initialized.

Java example: java/programs/control/DataFlow.java