Three Important Words

Program: A text in a “programming language” describing a sequence of state transformations.

Module: A collection of programs intended as a “work assignment” for designers or maintainers.

Process: An activity at run-time. A sequence of events that “takes turn” or “shares a processor with” other processes.

What is a Programming Language?

1) A programming language is a notation for describing *sequences* of state changes.
   - Individual Finite state machines are sequential.
   - Networks can be described as non-deterministic machines.
   - Not much happens in one event (state change).
2) Think of the "program" in a TV Guide:
   - sequence of events, but each is a program.
   - multi-level sequence of sequences.
   - conditional deviations for special events, time-zones, Newfoundland.
3) Building a program is a three-stage process:
   1. Describe a representation of the states of the machine.
   2. Describe the initial state.
   3. Describe state-transition sequence(s)

The “two machine” Machine

It is conventional to treat a computer as comprising:

- a program machine, and
- a data machine.

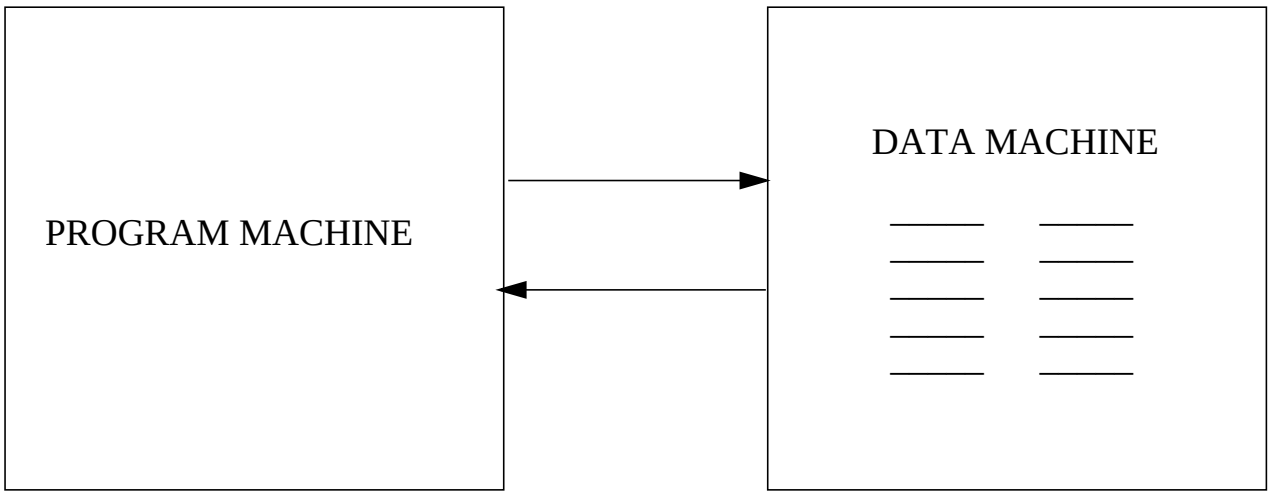
The data machine is (usually) much larger.

The data machine is repetitive, i.e. regular.

The program machine is initialised with a “program” (the program machine’s data).

The data machine is a set of smaller machines, called variables.

Data machine variables are not the same as mathematical variables.



Programming Variables

They are no different from the FSMs we have discussed.

They respond to inputs by changing state, giving outputs.

Mathematical variables are simply notational devices - strings appearing in function descriptions.

To avoid confusion we often refer to programming variables as *objects*.

The Data Machine

Basic *program variables* that constitute the data machine are simple.

Their input alphabet is just like their state set.

The NS function table has identical columns, each input leads to the corresponding state.

The output alphabet is the same as the input alphabet.

The output is the state.

NS	1	2	3	4	5	.....	N
1	1	2	3	4	5	.....	N
2	1	2	3	4	5	.....	N
3	1	2	3	4	5	.....	N
4	1	2	3	4	5	.....	N
5	1	2	3	4	5	.....	N
...							
N	1	2	3	4	5	.....	N

OUT	1	2	3	4	5	.....	N
1	1	2	3	4	5	.....	N
2	1	2	3	4	5	.....	N
3	1	2	3	4	5	.....	N
4	1	2	3	4	5	.....	N
5	1	2	3	4	5	.....	N
N	N	N	N	N	N		N

The Program Machine

The program machine controls the input to the variables.

The state changes in the program are determined by variable values (states).

We think of the program as instructions for state changes in the data machine.

Every programming language has four components

(1) Facilities for describing the objects that make up the data state

(2) A set of primitive programs

(3) Facilities for constructing bigger programs from primitive programs and previously constructed programs.

(4) Facilities for naming objects and programs.

Primitive Programs can always be added.

New data types can be created using old ones.

The essential parts of a programming language are the naming and construction facilities.

Types of Variables

Most primitive programs have the following form:

<variable identifier> ⇐ <expression>.

usually called an “assignment statement”

The expression (term) can be evaluated to something denoting a possible state of the variable.

The variable takes this as input and changes state.

Variables come in classes, called *types*, which are characterised by the answers to the following questions.

1) What is the set of states (values)?

2) What relations/functions are defined on those sets?

Boolean Variables

Two states, called {0,1} or {***false***, ***true***}

"=" maps pairs of booleans to booleans
"$\neg$" maps booleans to booleans
"$\wedge$" maps pairs of booleans to booleans
"$\vee$" maps pairs of booleans to booleans
"$\Rightarrow$" maps pairs of booleans to booleans

These operators have the meanings used in predicate logic.

Integer Variables

States {-N,-N+1, ... 0, 1, ... M}
where N, M > 0 and usually “close”

(Note that the above is a finite set)

“=” maps pairs of integers to booleans
“+” maps pairs of integers to integers
“-” maps pairs of integers to integers
“/” maps pairs of integers to integers
“÷” maps pairs of integers to approx. reals
“>” maps pairs of integers to booleans
“<” maps pairs of integers to booleans

Approximate Real Variables

It is impossible to represent all the reals, even in a given range.

The range is usually divided up into “equivalence areas”.

A representative is chosen for each area. Computation is done using the representative.

Often called “floating point” numbers.

<mantissa> <base> <exponent>

If a, b are representatives, a+b, a-b, a/b, etc. may not be represented. Errors result.

The choice of the areas and the representatives is very difficult.

Bad choices lead to inaccurate computations.

More discussion later in the programme

Program Variables, Identifiers

Every program variable is a finite state machine.

We give each program variable a name, e.g. “x”, “y”, “z”, “Rockin”, “Bob” “height”.....

Don't confuse the program variable and its name (a.k.a. identifier).

In mathematics, x is a variable, but in programming, “x” is the *name* of a program variable.

You write the names in the programs, never see the variables.

We sometimes call “x” a “variable” - it is sloppy. We should talk about “the variable identified by x”.

Constants

A constant is a string representing one of the possible states of a type of variable.

It is considered to belong to that type.

The identifier often tells us which state is represented. (e.g “23”)

Constants may be used as if they were variables in expressions. They will have the same effect as a variable in the associated state.

Constants need not be used in the data state representation. We only represent those things that change.

Three Classes Of Programs

1) Always Terminating
2) Never Terminating
3) Conditionally Terminating.

Each program determines possible sequences of state changes:

1) there is always a final state.
2) the sequence is always infinite.
3) sometimes there is a final state sometimes the sequence is infinite.

All three classes are useful.

In this class we will deal only with classes 1) and 3) and only be concerned about the finite behaviour of 3).

Other behaviour relevant to “real-time” programming, and Software Design III.

Primitive Programs

We build programs by **constructing** them from smaller programs.

Some programs must be “primitive” - basic building blocks.

We need two kinds of primitive programs

(1) programs that evaluate expressions in a given data state, i.e. $x+y$, x^2, $x > 0$,

(2) programs that change the state of one or more program variables. These are called assignments.

a,b,c,d, ... ⇐ <expr>,<expr>,<expr>,<expr> .....

Expressions may be thought of as assignments to unnamed variables, which we will identify by #, $\#_1$, $\#_2$, $\#_3$, ...

Program State vs. Data State

The distinction between program state and data state is arbitrary.

Information in the program can often be moved into data.

Information in the (initial) data could be program.

You can use this to great advantage when writing programs.

Programs can modify themselves.

The latter is not permitted by most languages.

The latter will not be done in this course.

When you program in assembly language you can write programs that modify themselves. This often leads to errors that are hard to understand.

Declarations: Designing the Data Machine

The first step in writing a program will be to describe the data machine.

We do this by declaring what variables (objects) we want.

Some languages allow declaration by first use. This leads to subtle, hard-to-detect, errors.

The data representation (the set of variables) is often the most important programming decision.

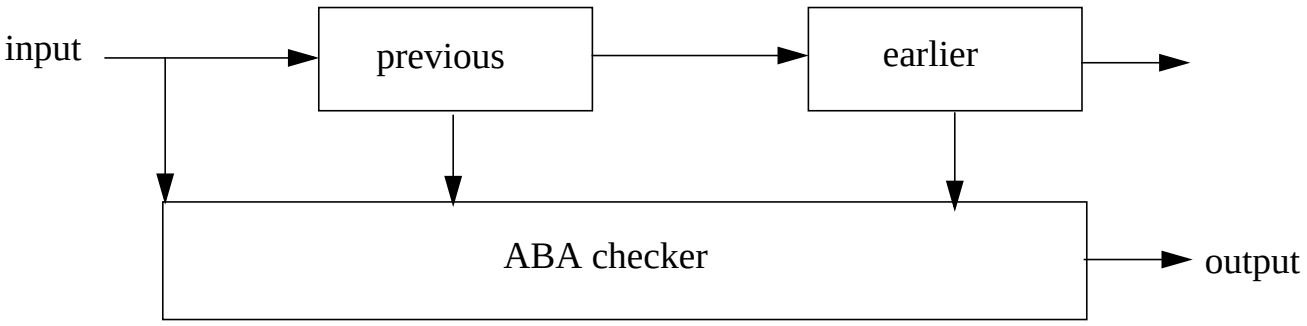
Unless care is taken, it is the hardest decision to reverse.

"*Show me your algorithms and I will ask to see your data structures; show me your data structures and I may not need to see your algorithms.*" F. P Brooks, Jr. - IBM/UNC

Designing a Data Structure

Designing a data structure is designing a network of FSMs

Consider (once more) an ABA recognizer.



The top two machines will be simple variables, with states corresponding to the possible inputs

The ABA checker evaluates a simple boolean expression. It has no variables of its own.

The program must transfer the old “input” to “previous” and the old state of previous to “earlier”. Then it evaluates the expression.

Because the data structure is “right” the program is simple.

Summary

Programs may sometimes, always, or never terminate.

In basic programming classes, our focus is on programs that calculate values and quit.

The *first* step in programming is always picking a data structure.

The *most important* step in programming is picking a data structure.

A data structure is a set of simple finite state machines called *objects* or *variables*.

Constants, denote object states and may belong to more than one type.

Data structure *design* is language independent.

Data structure *declarations* are very language dependent.

Thinking about Large Programs

What is a "large program"?

- One too large to be written by one person in a few days.

Why are large programs different?

- There are many tiny details.
- It is difficult, often impossible to keep all of those details in mind all the time.
- When there are several programmers, nobody is familiar with everything in the program.
- If the program is written over a period of more than a few days, people forget the details of what they have done.
- Errors in one part often affect other parts.

How should we respond to these problems?

- We organise programs into *modules.*
- Production of a module is a piece of work for a programmer or a group of programmers.
- Modules can be subdivided into smaller modules.
- Even individuals organise their work in modules.

Is modularisation an engineering issue?

- Some treat it as management: assigning people work.
- There are technical issues: issues independent of the people available.

What is a module?

Historically modules were simply a unit of measure, e.g. *3.27 square meters.*

Manufacturers learned to build parts that were one unit large.

The word now means *the parts themselves.*

Modules are usually relatively self-contained systems that are combined to make a larger system.

Design is often the assembly of many previously designed modules.

The constraints on modules

If the modules are hardware, how you put them together is obvious; there are well-known physical constraints. There is a well-identified time at which modules are assembled to get the larger system.

If the modules are software, there are no obvious constraints. Theoretically software modules can be arbitrarily large, their interfaces arbitrarily complex.

During software development there are several different times at which parts are combined to form a whole.

During software development there are several different ways of putting parts together to get the whole.

This makes “module” a buzzword. You have to listen carefully to know what the other person means and explain carefully how you are using the word.

Here it is an assignment for you or your partners.

Modules of software--when are parts put together?

A. While writing software
   - parts: work assignments for programmer(s)
   - when: files containing programs combined before compilation or execution.

B. When "linking" object programs.
   - parts: separately assembled (compiled) programs with "relative" addressing
   - when: addresses are inserted to provide links before execution.

C. When running a program in limited memory.
   - parts: memory loads data and programs that refer to each other by memory addresses
   - when: while the program is running.

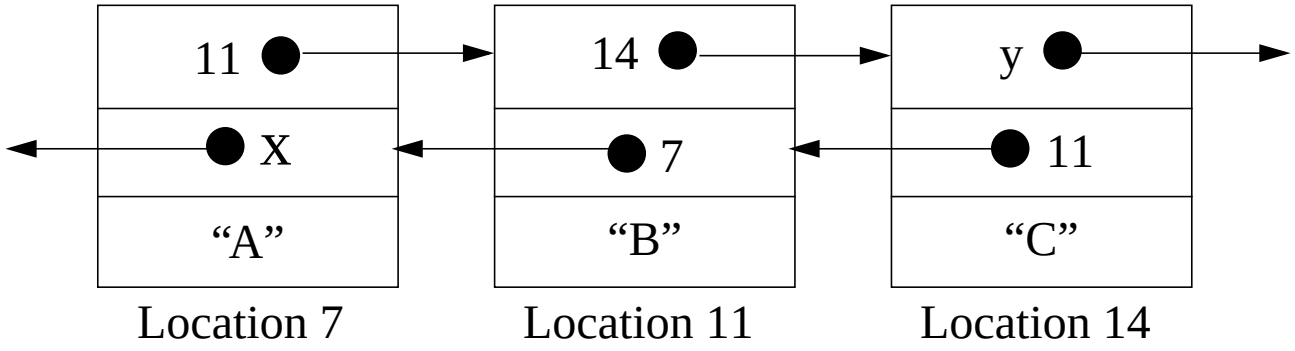
The word "module" is used in programming literature for all three of these!

The ambiguity in the word leads to confusion.

In this course, we talk only about the first meaning.

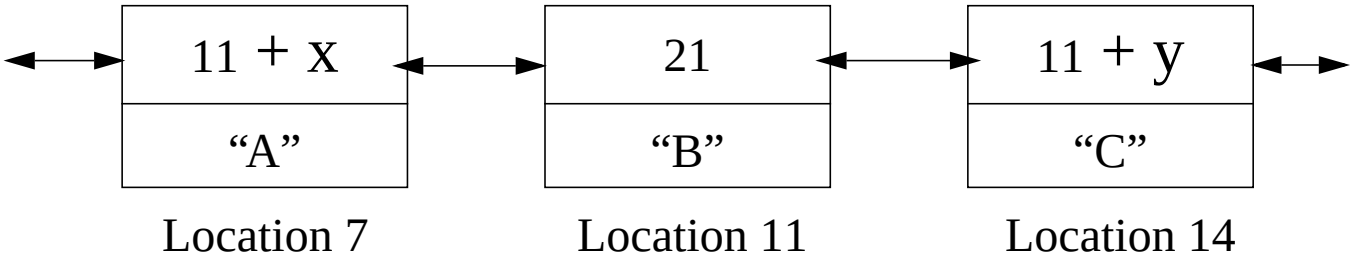
Example Linked Lists

Consider the following data structure:



This allows easy insertion and deletion.

Consider the following Alternative.



The link at each element is the sum of the forward and backward links.

Each alternative has advantages and disadvantages.

The decision is likely to change.

Example: Linked List, using information hiding

Make the job of maintaining the list a module.

Provide the following *interface* programs.

- Current: returns the value of the current element.
- Moveforward: current element becomes next on list.
- Movebackward: current becomes previous on list.
- Atstart?: returns **true** if you cannot move backward.
- Atend?: returns **true** if you cannot move forward.
- Insertafter(x): insert a new element in the list after current; the value of the new element will be x.
- Insertbefore(x): insert a new element in the list before current; the value of the new element will be x.
- Remove: Deletes the current element from the list.
- Alter(x) Changes the value of the current element to x.

Write all the programs that use this list using only the interface programs.

- Your programs can do everything that they could do if they used the link structure directly.
- Your programs will be easier to understand and change.
- Your programs are more likely to be correct.

That's why we group these programs into a module.

Dates should have been handled this way!

Another Problem: Need for Structure of Run-Time Events

Two ways to view events on a time-shared computer system:

Chaotic unrepeatable sequences

. . . read line typed on terminal by user A
fetch FORTRAN compiler into core for user B
compute value for user C
start compiling for user B
decode line typed by user A
output value computed for user C
respond to decoded command from user A . . .

Set of user jobs proceeding independently.

A	B
read line typed in decode line	fetch Fortran compiler into core
respond to decoded command . . .	start compiling . . .

The Problem: Need for Structure of Run-Time Events

Two ways to view a real-time system on a dedicated computer

1. First page from A-7 math flow

initialise navigation, if needed
calculate magnetic heading
calculate groundspeed and total velocity from inertial north and east velocities
determine whether aircraft airborne, land-based or sea-based
determine if inertial platform ready and reliable
format horizontal velocity and total velocity for panel
output zero to ground track needle
if ground align just selected, zero panel clock and turn on light
compute true heading . . .

2. Set of purposeful sequences such as:

set scale for inertial platform accelerometer pulse
read in accelerometer pulses and calculate N and E velocity
calculate inertial groundspeed from N and E velocities
damp inertial groundspeed with system doppler groundspeed

What is a Process

Processes are Subsets of the Events Occurring in a System

This is a definition, not a design criterion

In a sequential process, the events are fully ordered in time

This is a definition, not a design criterion

Almost any decomposition satisfies these definitions.

We need to agree on criteria

This will be discussed in SE III

Three Aspects of Process Design

A. Deciding on the right subsets (processes), i.e., grouping the events into processes.

B. How subsets cooperate and communicate.

C. Determining actual run-time order, i.e., scheduling,

- Determining when to schedule. (run time or earlier)
- Choosing a scheduling algorithm.

How Do These Three Concepts Fit Together?

Programs are always grouped into modules.

The programs within a module are designed together.

Processes are controlled by programs.

The process structure (division into processes) is never seen all at once.

Each module designer may decide how many processes to use.

Some modules are purely sequential. They are to be invoked by other programs.

Some modules are “persistent”. They have their own data and may be “running” while other things go on.

In the first two courses we ignore processes.

You will build purely sequential programs.

The important concepts for now are module and program.

Many people do not make this distinction carefully.

Beware of communication problems.