4. REQUIREMENTS ANALYSIS AND SPEC.

Last Mod: 2024-05-09

This section of the course will look at:

- Data, functional, and state models of a system.
- Distilling requirements into definitive forms.
- State machines
- Syntax diagrams
- BNF.
- A future lecture will cover the data model in more detail. ERD/ORD and normalization.

Exams:
cover definitions
Ask you to create diagrams
Ask about use-cases

Table of Contents

4.1 Required Readings

- Appendix A: Structured Analysis using DFDs
- Read lightly these 3 sections from this UML reference website.
  - Section 2 on Use Cases (but it doesn’t cover use case diagrams).
  - Section 5 on State Diagrams.
  - Section 6 on Activity Diagrams.

  https://cse.unl.edu/~goddard/Courses/CSCE310J/StandardHandouts/ShortUMLreference.pdf

4.2 Demarco’s Three Views

[Demarco82] suggests one needs to specify 3 independent views of a software system.

This is kind of an analogy to architect’s drawings for:

- front view,
- side view,
- floor plan.

Otherwise, the shape of a building can’t be *totally* defined for construction.

In computer software, the 3 views are:

- The **retained data model**. The specification of data (records/objects) retained in a system (either temporarily or longer).
  - I.e. What the system remembers!
- The **functional model**. How the input data gets routed, stored, and modified on its way to the outputs. Data-flow models
  - I.e. What the system does to the data!
- The **state transition model** for what should happen in each possible state for each possible event.

4.2.1 Retained Data Model

The **retained data model** shows the definition of records/objects and relationships between the objects.

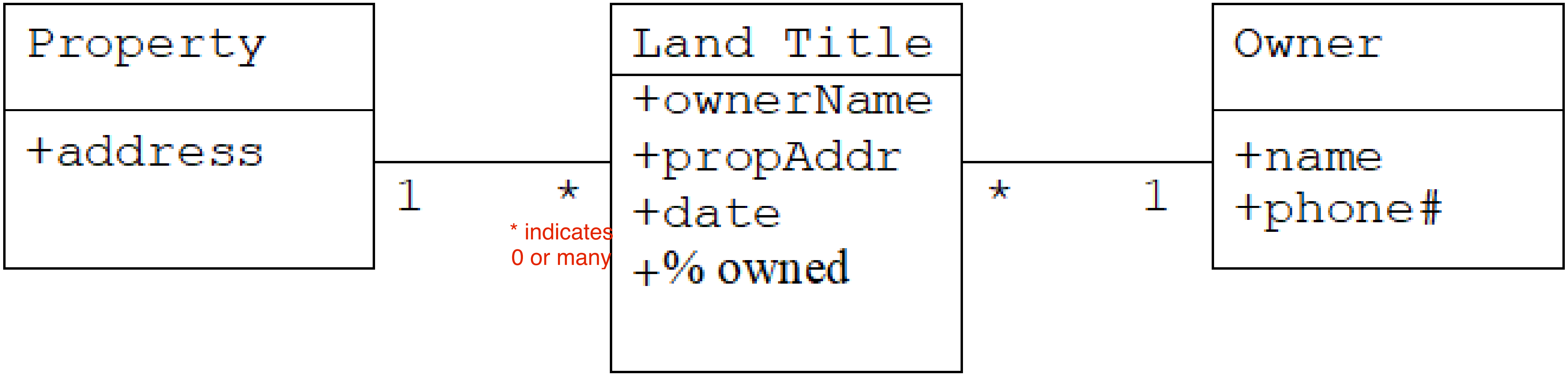
In database terminology, these are specified in an “Entity Relation Diagram” (ERD).

- In object-oriented UML, this is called an Object Relationship Diagram (ORD) or Class Diagram.

Such diagrams specify:

- What ‘knowledge’ (i.e. data fields) are retained by each object or record.
- And show the important and sometimes complex “relationships” between the data entities.

Very Simple Object Relationship Diagram



Property

+address

1

*

* indicates 0 or many

Land Title

+ownerName

+propAddr

+date

+% owned

*

1

Owner

+name

+phone#

Owning a property does not imply you live in that property

Interesting aspects:

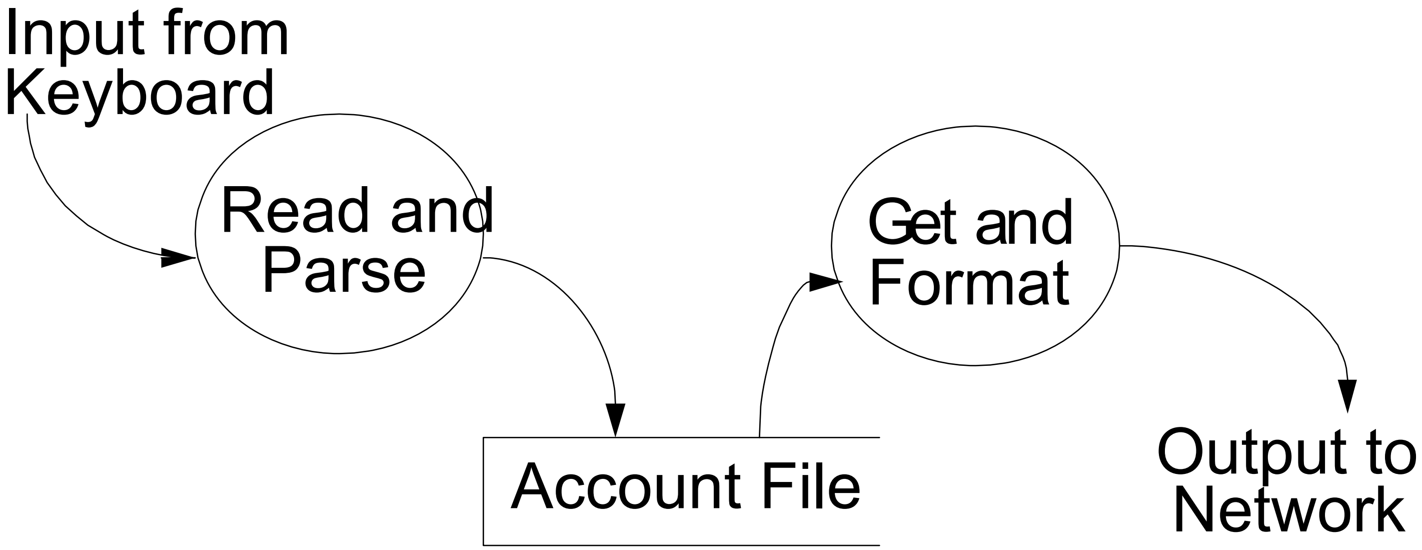
- After some analysis and normalization, this nicely handles shared property ownership.
- But should date be date_range?

If we wish to remember the previous owners, we must add it

- What if half the property is sold to a different owner?

4.2.2 *Functional/Data Flow Model*

The **functional model** specifies ‘where’ the data needed for the operations is to come from, and go to. Super simple example of Data Flow Diagram (DFD):



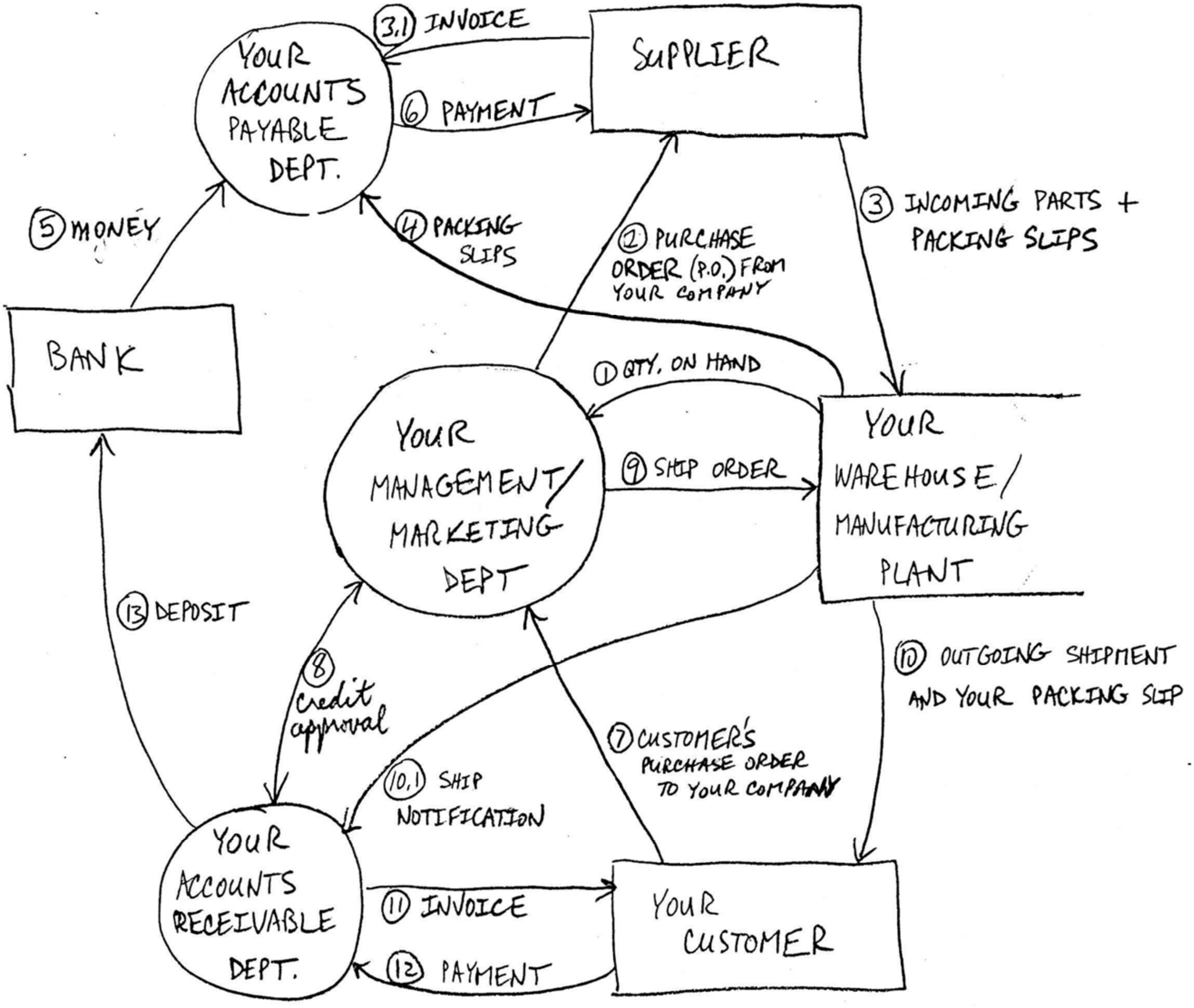
Understanding a business’s basic physical data flow, and terms like “Accounts Receivable”, helps you with your analysis.

Each business is different.

- A law office has customers but no warehouse.
- A manufacturing company typically has a warehouse. And a “resource planning” computer system, with product parts lists, inventory, parts ordering, product build work orders.

Here's a top-level, somewhat *general* business DFD.

Note the numbers on the arrows are for reference only, so a paragraph and specification could be written about each one.

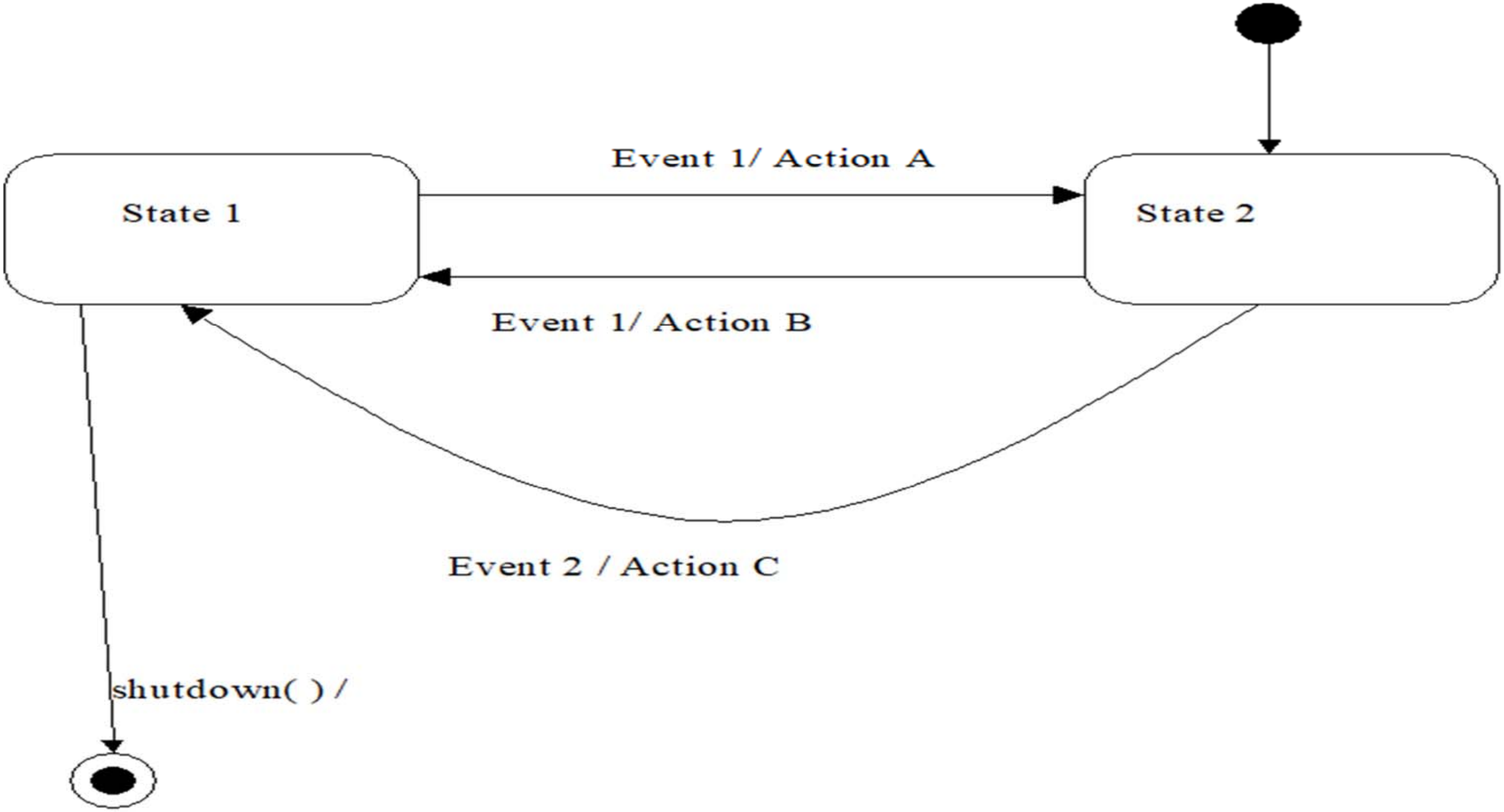

(3.1) INVOICE
SUPPLIER
YOUR ACCOUNTS PAYABLE DEPT.
(6) PAYMENT
(3) INCOMING PARTS + PACKING SLIPS
(5) MONEY
(4) PACKING SLIPS
(2) PURCHASE ORDER (P.O.) FROM YOUR COMPANY
BANK
(1) QTY. ON HAND
YOUR MANAGEMENT/ MARKETING DEPT
(9) SHIP ORDER
YOUR WAREHOUSE/ MANUFACTURING PLANT
(13) DEPOSIT
(8) credit approval
(10) OUTGOING SHIPMENT AND YOUR PACKING SLIP
(7) CUSTOMER'S PURCHASE ORDER TO YOUR COMPANY
(10.1) SHIP NOTIFICATION
YOUR ACCOUNTS RECEIVABLE DEPT.
(11) INVOICE
YOUR CUSTOMER
(12) PAYMENT

I will assign required readings for DFDs.

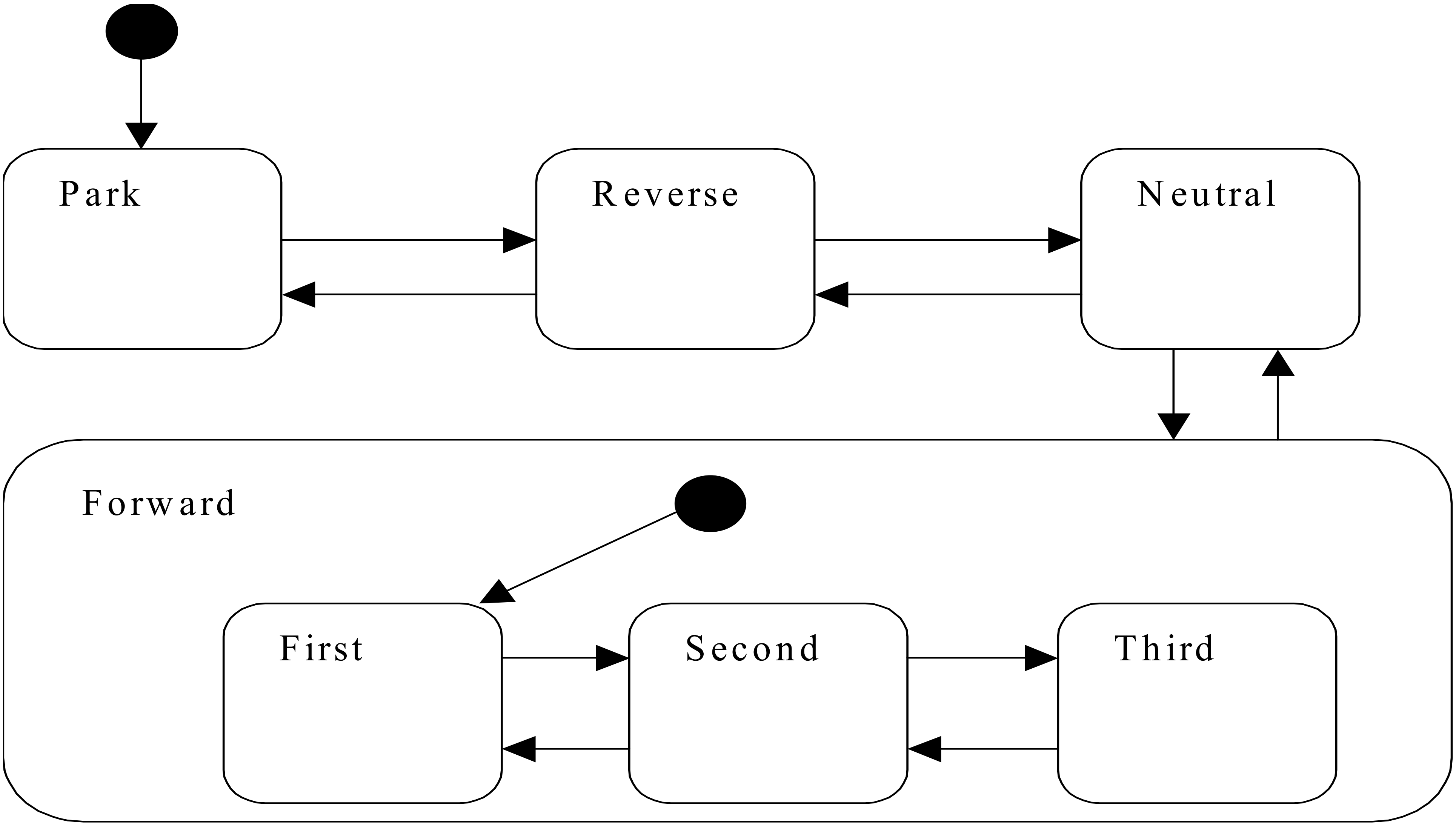
Then later in the course we’ll learn the object-oriented, modern UML versions called “Activity Diagrams”.

4.2.3 State Model

The **state model** specifies ‘when’ and ‘under what conditions’ (i.e. in which state transitions) various operations are invoked.


Event 1/ Action A
State 1
State 2
Event 1/ Action B
Event 2 / Action C
shutdown( ) /

Example of an automobile transmission with sub-states.



Substate machines are denoted by a box

4.3 Specification Rigor

There are 3 levels of specification rigor:

- Natural language.
- Diagrams and/or structured languages.
  - E.g. pseudo-code, tables, graphical diagrams.

- Or, using “formal techniques” which are absolutely definitive and rigorous. e.g.
  - regular expressions using BNF
  - syntax diagrams
  - finite state machines
  - relational algebra
  - VDM or Z
  - axiomatic specifications
  - algebraic specification of abstract data types.

We’ll cover only a few of these formal techniques.

4.4 Noun-Verb Analysis

One way to find entities/records/objects is from the description of the proposed system written by the customer. Generally:

- many (but not all) nouns will turn out to be software objects/records.
- many (but not all) verbs will turn out to be operations/functions/processes.
- adjectives and adverbs may add contributory information about the objects or functions.

- possessive tense is often an indicator of an object’s attributes (e.g. the car’s licence number.).
  - Alternatively, possessive tense may helpfully indicate a relationship (e.g. vehicle’s owner, where owner and the possibly many vehicle’s she owns are two different but related object classes).

4.5 Behavioral Modeling

Data modeling provides only a very static view of the system to be developed. Before starting to design or code, it is also important to gain an understanding of the desired behavior of the system and of each command over time (i.e. the sequence of things that should happen automatically and in what order in the program, once you start a complex command). This is called ‘behavioral modeling’ because we gather (or hypothesize based on our knowledge of the requirements) the necessary sequencing and present them using some kind of (usually) diagrammatic model.

The purpose of documenting the behavioral model is so:

- customers can review the sequencing for correctness.
- the sequencing specification can tell the designers and programmers what to design.
- An existing or draft user manual can either be:
  - the source of the externally visible behavioral model.
  - Or he behavioral model can help specify the various steps to be shown in the user manual.
- the behavioral model can provide the information needed by the test department to start writing tests cases for the test plan.

It is only in the late ‘90s that behavioral modeling using the concept of ‘use case scenarios’ had been appreciated as being so important.

- You can read ahead in Section 5 of the lecture notes to see how we will later use ‘use case scenarios’ in both our external and internal design processes.

In the meantime, during requirements gathering and analysis it’s important to ask users/customers what events/steps/screens they expect to happen and see. *This is very important to your understanding and analysis of the requirements*.

Some events will be external, like a bank receiving a deposit.

Others will be internal like ‘after the user selects ‘install’, (s)he will see this, then be given a choice about that, then will hear disk noises, then will see a “the application has been installed” message’.

By inquiring of the user what (s)he expects, you might learn about some feature you didn’t know the customer needed, or about some special sequence of events, or particular step in a sequence, that you hadn’t previously heard of.

Asking about events and sequencing is another way to get the user/customer to do the talking.

4.6 Finite State Machines

You’ve likely studied finite state machines (FSMs) in MACM 101 and ENSC 252.

State machines are wonderful (!) formal/definitive methods for specifying the actions to be taken circumstances (certain states/modes).

- What should an airplane do when asked to retract it’s landing gear when not in the “flying” state?

In addition:

- They are finite!
- And hierarchically decomposible.

Easy to determine how many scenarios to test.
That is what is meant by easy to test

Thus easy to specify, program, and test.

- They are exhaustively testable!

And they're absolutely definitive if properly specified.

Many objects that software components model have context sensitive behavior:

- You shouldn’t heat a milk tank if it is in the empty state.
- You shouldn’t create a diploma if a student hasn’t reached the graduated state.
- You shouldn’t pop from a stack that is empty.
- Network connection objects often react to incoming packets in different ways depending on their current state.

Not all reactive objects are state machines.

- If they react the same way every time you call the same member function, then they are not really a state machine (or can be regarded as a state machine which has only one state).
- Only if an object is to react differently TO THE SAME EVENT, in different historical contexts (i.e. remembered state), is it a real FSM.

State machines are very common.

When you press the SHIFT key and then the “a” key, you get an upper case “A”.

- The FSM remembers (using a state variable) the original press of the SHIFT key thousands of microseconds ago, that it is in the upper case state.
- When “a” is pressed, the code refers to the state variable to determine whether an “a” or “A” is being entered.

A reactive component’s state is a **remembrance** of **historical context**.

The state may have another name. E.g. current “status” or “mode” of the component.

Past (i.e. historical) events have caused the component to change into the current mode.

E.g. In in MS-Windows, typing a **`<shift>-a`** actually causes 4 event messages to be sent to your program:

1) the <shift> key is pressed.

2) the ‘a’ key is pressed.

3) the ‘a’ key is released.

4) the <shift> key is released.

FSMs can also be very complicated.

The state table for the behavior of a network driver communicating with another computer using the HDLC protocol is 23 pages long.

Like all state tables, it basically says:

- If I have just sent an acknowledge packet (i.e. am in acknowledged state),
- and I receive a negative acknowledgement packet,
- then I should send some other kind of packet and likely adjust some variables,
- and change to some new state,
- then wait for the next event.

Students have problems with state machines because they are not like a flow chart.

They do not just have one or two paths through them.

A complicated state machine shows every possible path taken in every possible combination of circumstances.

It’s like thinking of every possible route to Richmond (including via the Mission bridge!) at once.

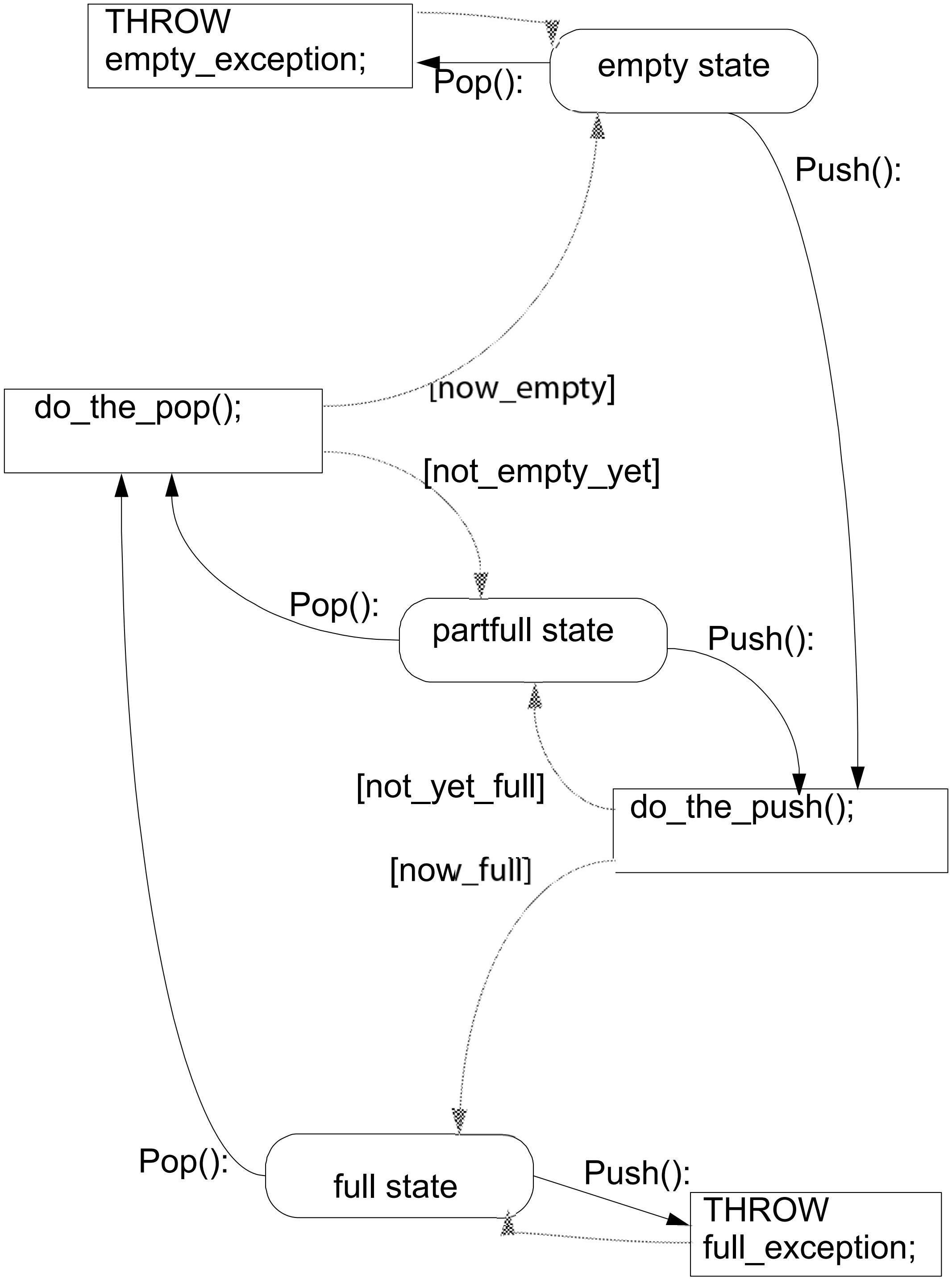
- [Though routing to Richmond is not an FSM] The best way to handle FSM complexity is one scenario/path (sequence of common steps) at a time.

The complete FSM (map?) is then the union of all the scenarios.

Let’s elaborate on a simple stack example to see the proper way to program context-sensitive, software components.

Finite State machines can be documented either with state transition/action diagrams, with tables, or with pseudo-code.

The diagram is the most intuitively appealing, so I will show you it first.


THROW
empty_exception;
Pop():
empty state
Push():
do_the_pop();
[now_empty]
[not_empty_yet]
Pop():
partfull state
Push():
[not_yet_full]
do_the_push();
[now_full]
Pop():
full state
Push():
THROW
full_exception;

The ovals represent the states.

- They represent the appropriate subset of the memory of all possible past history.

A finite state machine doesn’t remember everything that’s happened to it.

- So it remembers an appropriate finite subset.

The bold arrows represent events that can happen to the FSM.

- In software, these are usually procedure calls to the component modeling the FSM.

Boxes represent behavioral actions that are executed by the FSM on its way to its next state.

The dim arrows indicate which state comes next.

- Most FSM techniques do not use dim arrows guarded with exit conditions.

  Are the exit conditions the requirements that need to occur in order to transition into the next state?
- But I like them as they frequently reduce the number of rectangular boxes and arrows needed to fully document the behavior.

The exit condition is needed when you have the same event mapping to different states on the same state.
It is a way to create states which are mutually exclusive

Note that when you call a function, the action specified by the name of the procedure should **not** be viewed as always being done.

- Design software components so if it’s inappropriate in that state to fulfill the ‘request’, the function will not do so.
- This is why we say objects have ‘intelligence’.

Now, how to properly program software components that have state.

First, you need to define the essential states.

Second, export function signatures for each named event/request/message that the component must handle.

Then put a switch statement as the outer block of each exported function implementation.

This is talking about how to approach software components with state using C++ or Java.
Why are we not doing a general analysis?
Russel answer: I am not doing a general analysis because I do not wish to.

```
Class Stack{
   enum StateType{empty, partfull, full};
   StateType state;  //initially empty.
public:
   Value Pop(){  //public request for a pop.
      switch(state){
         case empty: //leave state same + deny request.
                    RAISE empty_exception;
                    break;
         case partfull:  do_the_pop();
                    if (now_empty) state:= empty
                    else state:= partfull;
                    break;
         case full:          do_the_pop();
                    state:= partfull;
                    break;
      };
   }; //end Pop().
```

Could we not have one and the other case?
If it is not empty, then you can remove an element.
Then you need to adjust the state and return the removed element

```
   void Push(){  //public req for push.
      switch (state){
         case empty:           do_the_push();
                    state:= partfull;
                    break;
         case partfull:     do_the_push();
                    if (now_full) state:=full
                    else state:=partfull;
                    break;
         case full: RAISE full_exception; //deny req.
                    break;
      };
   };//end Push().
/*----------------------------------------------------*/
private:
   void do_the_pop(){  ... //actually do the pop.
   };
   void do_the_push(){ ... //actually do the push.
   };
};//end class
```

Note the lovely manner each exported function is implemented.

Why is this code lovely?

- Each begins with a switch statement which determines whether the software abstraction will honor the request, given the stack's current state.
- The actual pushing and popping is done in non-exported procedures near the bottom of the module.

If you write code this organized for most of your life, I will be very proud of you.

- [Tell story re was asked to assist a LAN driver project.]

Advice: If documenting a finite state machine using a diagram, it's very easy to forget to document what to do if in state 3 and event B happens.

- Best practice: Translate the diagram to a state table to see if unintentionally *blank table entries*.

Example State Table:

Memorize the format of this table

Event	State	Action	Next State
A()	1	Call sin()	B
A()	2	nothing	A
B()	1	Call cos()	B
B()	2	nothing	A
B()	3	Call tan()	B

The states here are incorrect.
They should be either 1, 2, 3.

Q: What’s wrong with the table above?

A: No row for event A() in state 3.

Or if there was a row for it, either or both the action and next state could be blank!

➔ Go back to customer: Ask what should be done for that event in that state!

In future lectures we will look at UML's FSM diagramming notation. And discuss other details.

For the first assignment, if you need to document an FSM, just use a table.

For the first assignment, you may need to do a state diagram instead of the state table

Optionally, look at the basics from the UML Poster [Rational], which illustrates:

- Start icon
- State icon – illustrates the most general stuff that can be shown in a state.
  - What to do on entry,
  - what to do while there,
  - what to do if some event happens that should cause an action but not change the state, and
  - what to do on exit.
- Transitions, including those that have mutually exclusive guards.

  - There can be two transition arrows out of a state cause by the same event, but ONLY if they are guarded by mutually exclusive conditions.

- Hierarchical nesting of states.
- History icon indicates that when you come back into a superstate, the sub-machine restarts in the last state it was left in, rather than in its start substate.

What is a superstate?

Look in the notes (20 slides back) and
in the reference notes

4.7 Variable Sequence Specification

It is possible to use Context-Free Grammars to specify the allowable sequences of the input to a computer application.

E.g. Telephone system must accept all variety of numbers such as:

291-4310

1-291-430

1-604-291-4310

Grammars are used to narrow the allowable input down to something somewhat variable in format, but which the system will understand.

Finite state machines, or parser, can be written to

- recognize an allowable sequence in the "language", or
- reject it as junk not following the grammar!

Provides for specification of rules for sequencing of almost anything (e.g. network packet types).

Grammars can be recursive. Think programming language "if statement" within an outer "if statement".

There’s no use specifying a grammar unless:

- Software is going to have to parse the ordering as part of:
  - Checking user input.
  - Determining the meaning of a string or sentence.
  - Finding and extracting part of a string.
- Or it will help you write instructions for a user manual for data entry by humans or a network.

The two methods to formally specify a grammar are Syntax Diagrams and BNF textural specifications.

Text

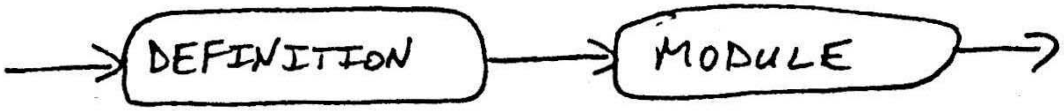
4.7.1 Sequence Diagrams

Must be able to draw sequence diagrams

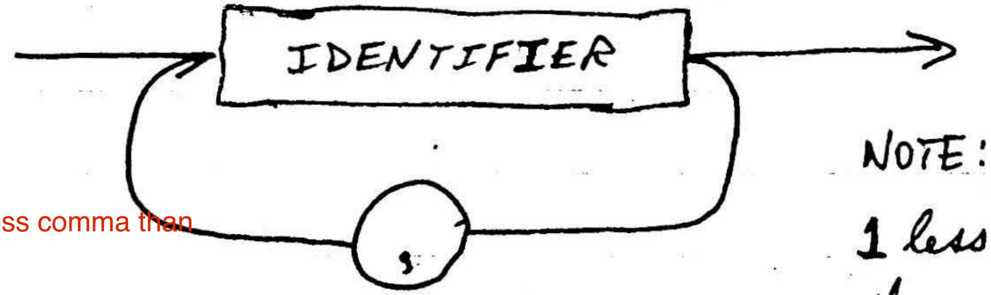
THE MAJOR ELEMENTS OF SPECIFICATION OF LEGAL ORDERINGS ARE:

Things in rounded ovals are atoms

SEQUENCE:



REPETITION:

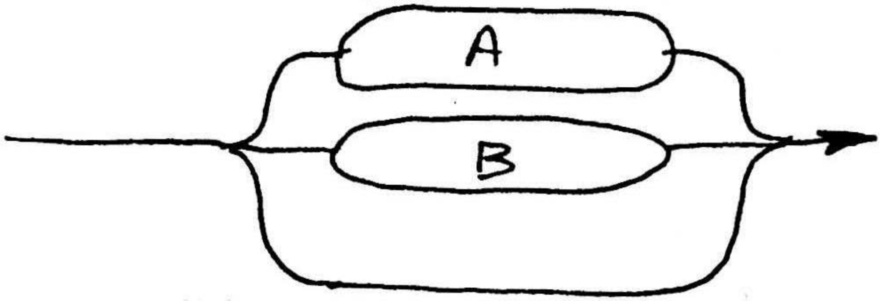


For this example of repitition, there will be one less comma than identifiers

NOTE: SPECIFIES 1 less comma than identifiers.

c.f. Natural language description of this may be ambiguous or wordy.

ALTERNATION:



(ALSO CALLED SELECTION)

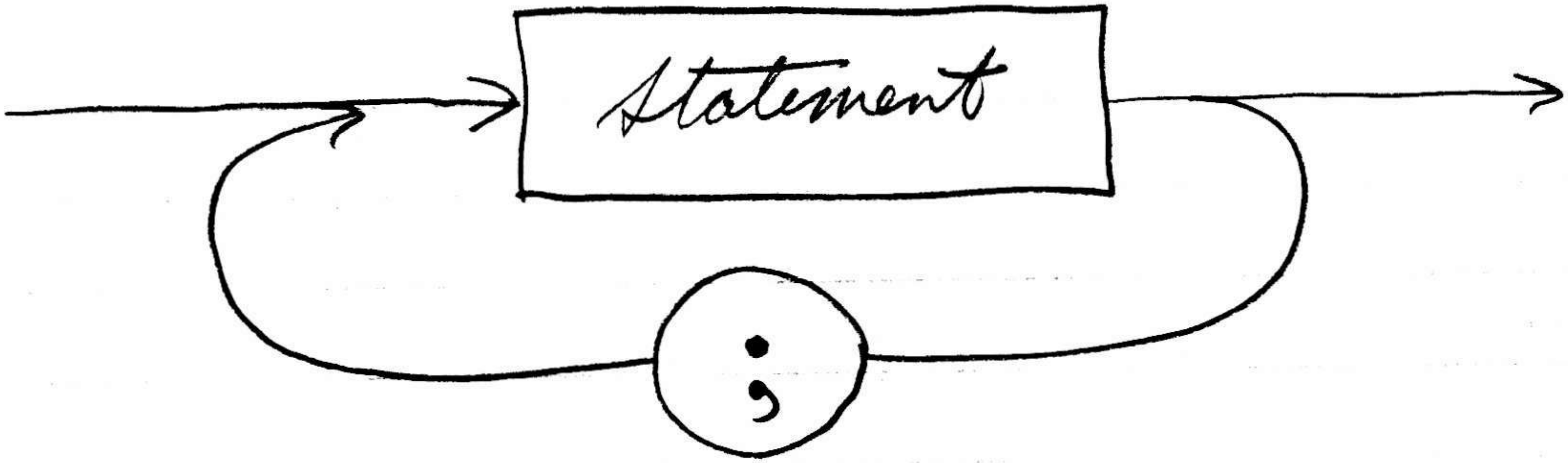
Ovals represent “atoms”.

Rectangles represent “composites”.

- Composites allow hierarchical abstraction. They indicate to see elsewhere for further refinement (sub-defining).

ANY GOOD INTRODUCTORY PROGRAMMING LANGUAGE TEXT WILL HAVE SYNTAX DIAGRAMS (OR BNF) FOR ITS LANGUAGE.

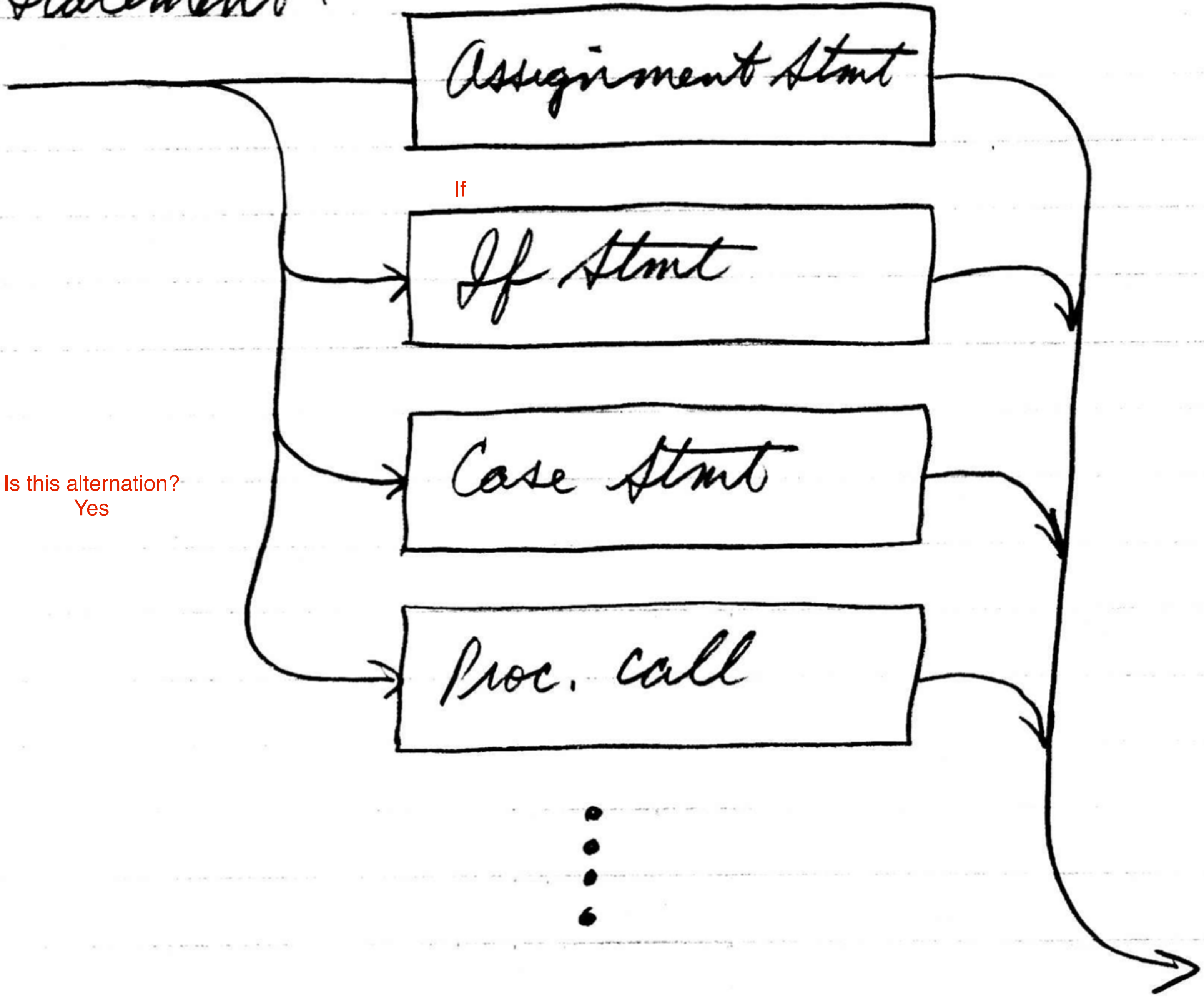
eg. Statement Sequence:



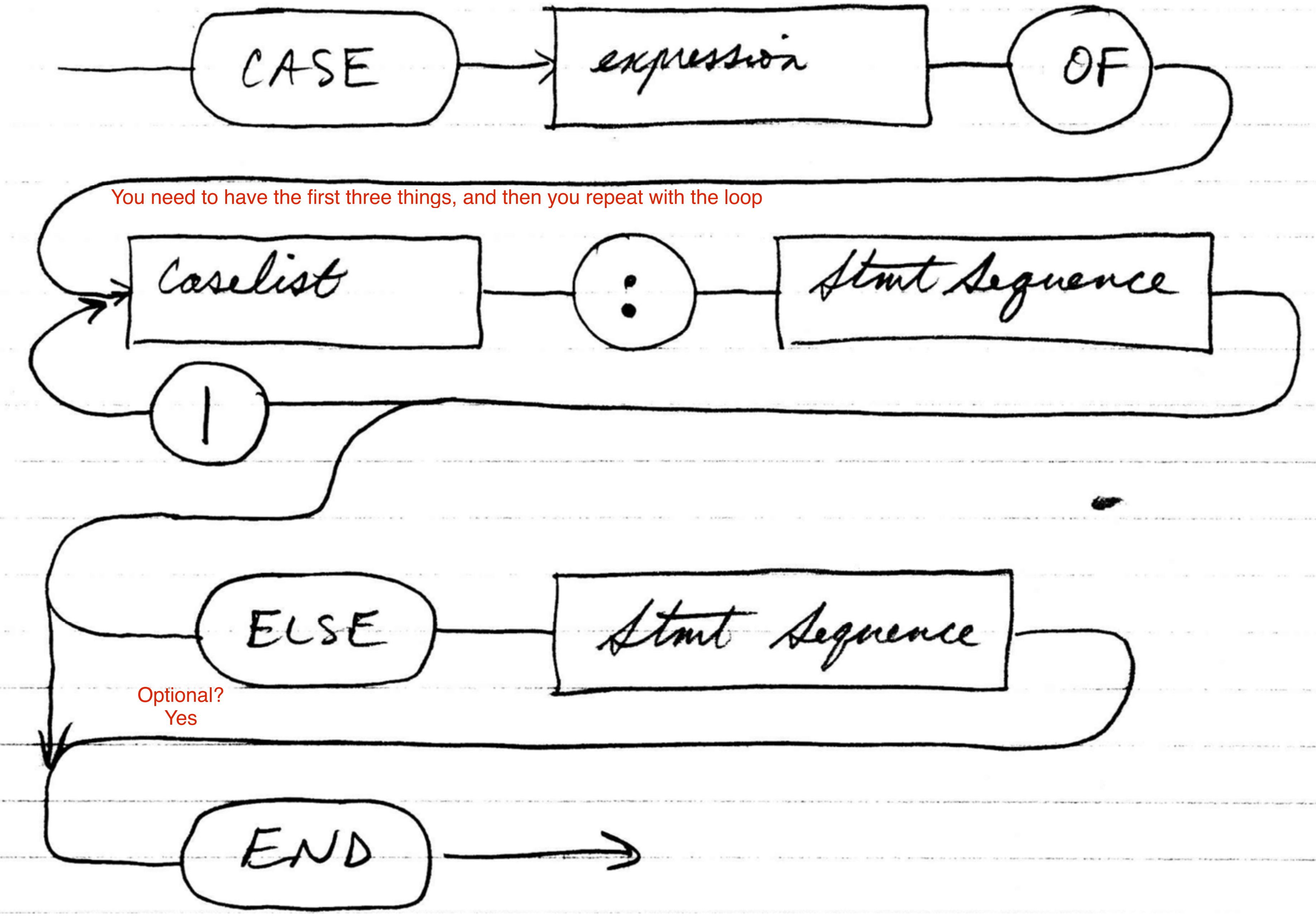
(This particular grammar indicates no terminating ':'. I.e. Is not C++.)

Does he mean semicolon?

e.g. Statement:



e.g. Case Stmt



You need to have the first three things, and then you repeat with the loop

Optional?
Yes

Note this shows ‘indirect’ recursion elegantly expressed:

- Statement sequence is made up of statements.
- Statement can be a **`case`** statement.
- A **`case`** statement clauses can be made up of statement sequences.

Syntax diagrams are excellent because humans are very good with visual reasoning.

Also, they help with:

- Completeness: All valid sequences are recognized.
- Closure: No invalid sequences are accepted.
- Correctness: No valid sequence is incorrectly interpreted out of sequence.
- Un-ambiguity: No sequence has more than one interpretation in a current context.
- Consistency: No two sequences have same interpretation (unless desired).

4.7.2 <u>Bacus-Naur Form (BNF)</u>

We can communicate same info as syntax diagram in written form.

This allows potential entry of a grammar into a compiler compiler!

Of course also good for denoting legal sequences in a Requirements Spec.

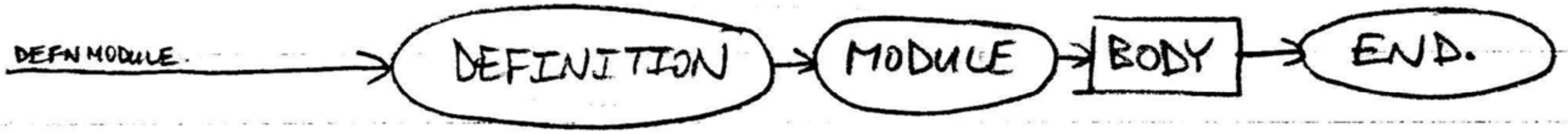
We just need:

- Sequence.
- Repetition
- Alternation.

A <u>SEQUENCE</u> IS COMPOSED OF A SERIES OF ATOMS, OR OTHER COMPOSITES.

e.g.
DEFNMODULE = "DEFINITION" "MODULE" BODY "END."

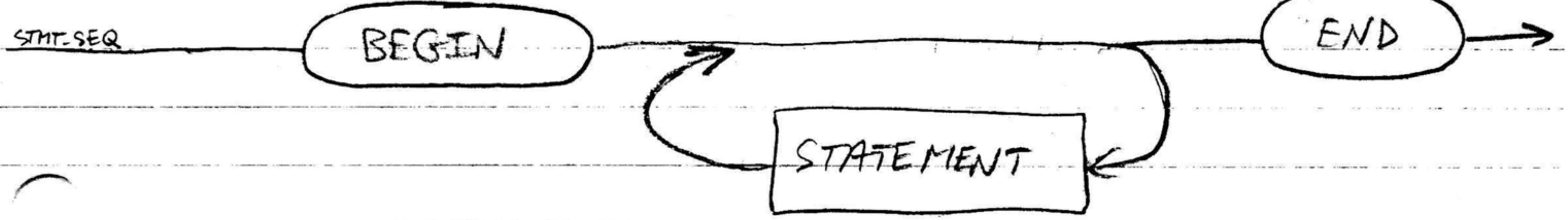
is same as:



<u>REPETITION</u> USES CURLY BRACES.

e.g. STATEMENT_SEQUENCE = "BEGIN" {STATEMENT} "END"

STATEMENT_SEQUENCE :



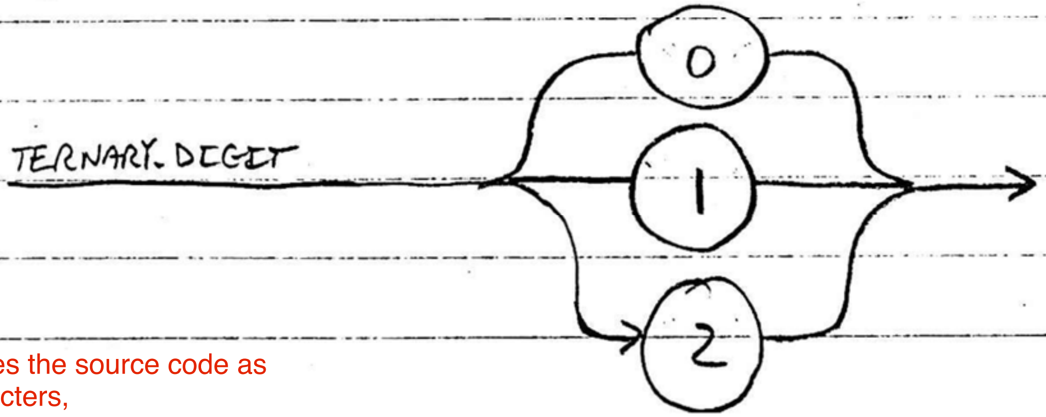
NOTE: { } MEANS "Ø OR MORE OCCURENCES OF"

$\{\ \}^5$ MEANS "0 TO 5 OCCURENCES OF"

ALTERNATION USES THE VERTICAL BAR "|" IN ROUND BRACKETS.

e.g. ternary_digit = ("0"|"1"|"2")

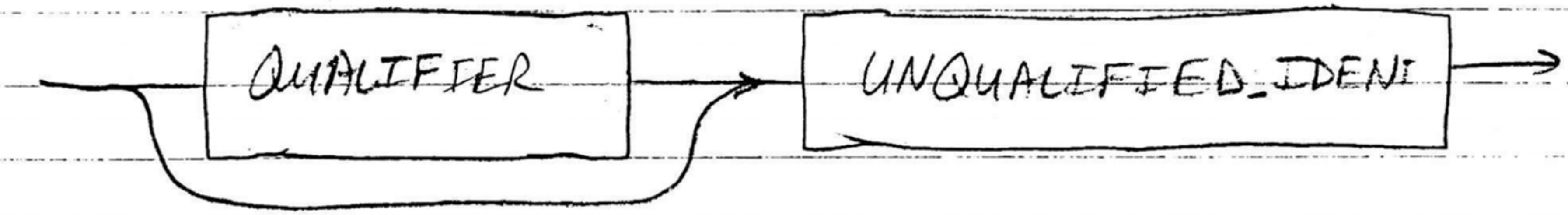
is same as:



If the parser consumes the source code as characters,
What happens when the compiler reads the assignment of an integer variable?
If the compiler reads this, how does it know if you assigned a string or an integer?

THERE IS ONE ADDITIONAL DENOTATION THAT IS SOMETIMES USED AND THAT IS:
OPTION USES SQUARE BRACKETS

eg IDENTIFIER = [QUALIFIER .] UNQUALIFIED_IDENT



OBVIOUSLY [ ] IS EQUIVALENT TO { }^{1},
AND MEANS "ZERO OR ONE OF".

Composites: Normally there is a way to denote composites from atoms in BNF.

I will require you to double quote atoms. E.g. fixed keyword strings in a programming language.

“if” “then” “else” “for”

This agrees more with some later courses and textbooks (other than previous Cmpt 276 textbook [Pressman97]).

In some other BNF dialects angle brackets < > are put around composites.

Example composite definitions:

Pair = (“a” | “b”)(“c” | “d”)

- Allows ab, ad, bc, bd.   Not ca, db, etc.

Sandwich = “A” [“B”] “C”

- Allows ABC or AC.

Prefix = “A” {“B” “A”}

- Allows A, ABA, ABABA. Not AAB, ABAB.

Suffix = { (“A” | “B”) } “C”

- Allows C, AC, BC, AAC, ABC.
- NOTE: Some might abbreviate this to {“A” | “B”} “C”, but don’t do it on a Cmpt-276 test! Marks will be taken off
- Also don’t abbreviate [(X | Y)] to [X | Y].

EmptyOrNot = [(“A”|“B”) “C”]

- Allows null, AC or BC.

OneOrMore = digit {digit}

- Allows 1, 21, 1234, 98765, etc.

CampusPhone = (Extension | Outside# | “0”)

Extension = (“4000”| “4001”| “4002|…)

Outside# = “9” (Local# | LongDistance#)

LongDistance# = (“0”|“1”) [AreaCode] Local#

AreaCode = digit {digit}^{10}

Local# = digit digit digit digit {digit}^{3}

Exercises:

- Convert the above CampusPhone grammar into a syntax diagram.
- Translate the previous programming language subset syntax diagram into BNF.
- Which would be better for a design review with a customer: Syntax diagrams or BNF?

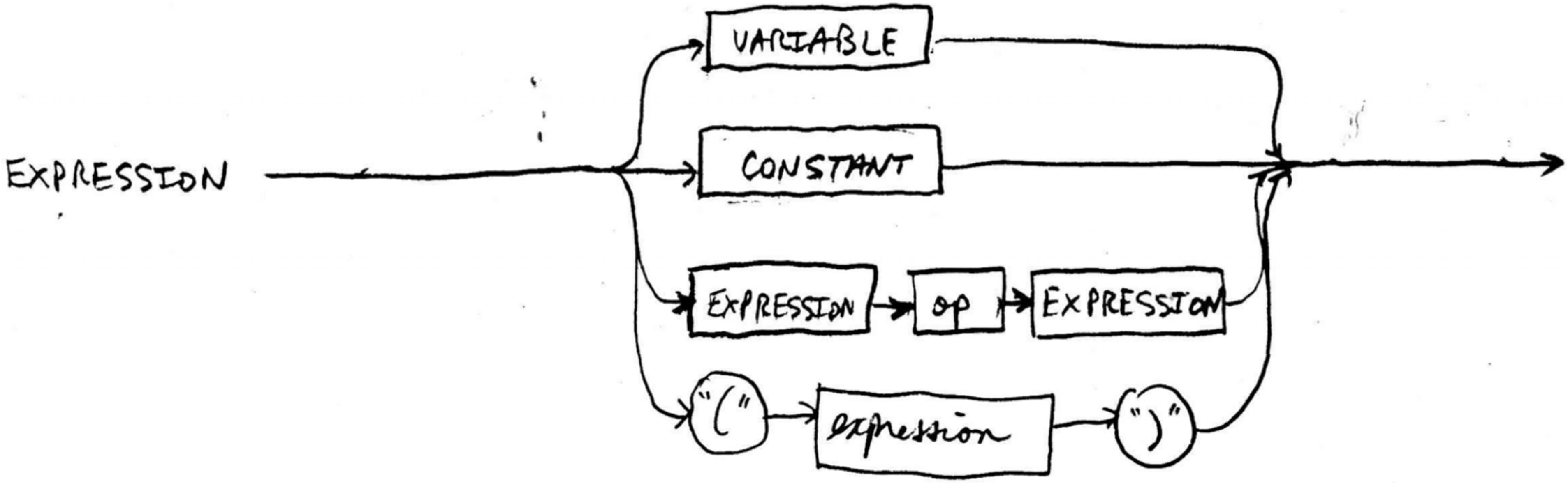
Uses for BNF or sequence diagrams:

- User operation sequence. E.g. Things must be done in a certain order before account s/w can generate a monthly balance.
- Because of certain inter-relationships of functions, they have to be called in a certain order. E.g. **`initialize()`**, **`operate()`**, **`clean-up()`**.
- Describing fields in a record.
  - Especially variable length records.
  - Or records with repeating groups.

Note #1:

BNF and syntax diagrams are good for expressing operation or definition recursion (if that’s needed).

E.g. expression = (variable|constant|
expression op expression|
"(" expression ")")



Note #2:

A BNF expression or syntax diagram is not necessarily unique. There may be several ways to express the same thing.

It is the same thing if there is no context to consider?

E.g. digit { digit } = { digit } digit

Note 3:

Can translate a syntax diagram into a state transition diagram (for the FSM that will recognize that syntax), change the syntax path intersections to states, and syntax ovals/rectangles to events.

4.8 Data Modeling and Normalization

Proper analysis is needed for an information system to store multiple tables of related information.

- Sometimes called ‘information engineering’.

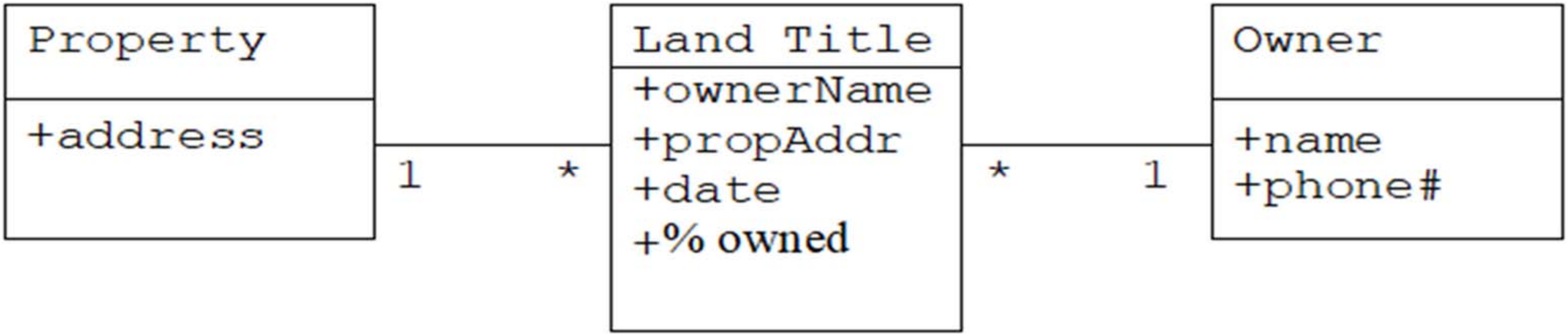
Note: SFU coops students have worked on huge projects that involved 1500 different types of records/tables.

- Each stored in a database management system (DBMS) in its own database table (i.e. file).

An important step in data organization refinement is called ‘normalization’.

- Result allows each table’s data records to be of similar length!
- Very important for fast, disk seek access.

Data modeling also ‘brings out’ many questions to ask the customer:



- Can a person own more than one property?
- Can a property be owned by >1 person?
- Can a property be owned by no one? Or a special entity called a municipality or province?
- If storing a municipality as an owner, does a municipality have different or more data fields than a person?

And, normalization results in a diagram where lines link to other record types.

This provides a map/mechanism to find related records in other tables.

- E.g. What course offerings is a student is enrolled?

Normalization is helpful even for the non-disk parts of an application. For example, object-oriented analysis and design.

OO analysis and design is a good because:

Why is functional programming not mentioned?

- It’s natural to the way humans think.
- Maps well to modular/ADT/OO design methods.
- Maps well to OO programming languages.
- Importantly, the resulting OO designs are more likely structurally stable over 10 years of maintenance.

How can you claim the designs are stable?
Is it grounded in a study?
Is it your personal opinion?

E.g. Though objects may need:

- a few new attributes or functions added or changed over time, or
- more or different objects

How do you know it will not likely change the majority of the code?

as the software is enhanced with new functionality, this will likely *not* require much change to the bulk of the design/code.

You'll unlikely, in future maintenance, need to cut objects in half and glue them together with other parts, because the objects are fundamental.

- E.g. For an airline, there will always be planes, flights, pilots, destinations, etc.

We’ll cover entities/objects, entity/object relationships, and normalization in a later section of the course.

4.9 References

[Booch94] “Object-Oriented Analysis and Design, 2nd ed.” by Grady Booch, Benjamin-Cummings, 1994.

[Coad91] “Object-Oriented Analysis” 2nd ed., by Peter Coad and Ed Yourdon, Prentice-Hall, 1991.

[Demarco 82] “Software Systems Development”, Tom Demarco, Prentice-Hall, New York, 1982.

[Jacobsen92] “Object-Oriented Software Engineering: A Use Case Driven Approach” by Ivar Jacobsen et al, Addison-Wesley, 1992.

[Kendall87] “Introduction to Systems Analysis and Design” by Penny Kendall, Wm. C. Brown Publishers, 1987.

[Kendall92] “Systems Analysis and Design” 2nd ed., by Ken Kendall and Julie Kendall, Prentice-Hall, 1992.

[Muller97] “Instance UML” by Pierre-Alain Muller, Wrox Press, 1997.

[Pressman97] “Software Engineering, a Practitioner’s Approach, 4th ed.”, by Roger Pressman, McGraw-Hill, 1997.

[Rumbaugh91] “Object Oriented Modeling and Design” by James Rumbaugh et al, Prentice Hall, 1991.

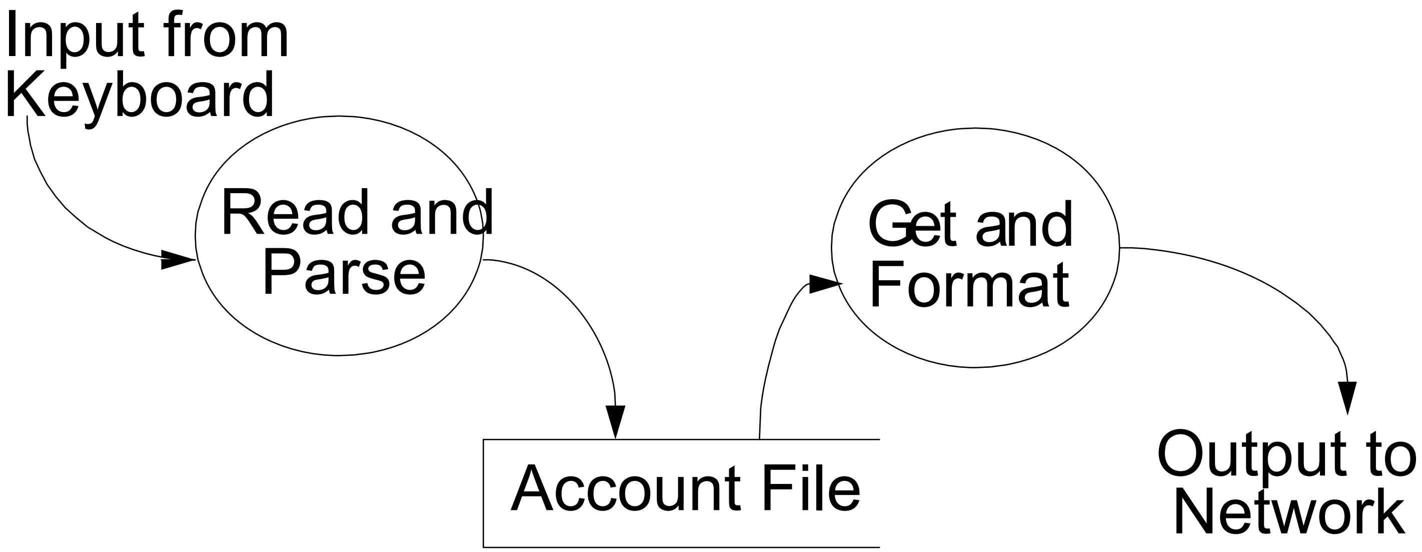
4.10 Appendix A: Structured Analysis using DFDs

We’re now going to look at the flow of data in a system. Some systems are highly flow oriented, and this can be documented by Data Flow Diagrams (DFDs).

The technique is called Structured Analysis and (Structured) Design (SAAD or SA/SD), is ‘somewhat static’ even though it seems to show data ‘flowing’ in a system.

But this flow is not actually sequenced; it just looks that way.

Let me illustrate while introducing the concept of a **Data Flow Diagram (DFD)**.



This diagram shows the essential elements in a DFD:

- input sources
- functional processes
- files

What is a functional process?

- output sinks
- though not shown above, you should also label the flow arrows with a name or description of the kind of data that flows along that particular arrow.

In the above diagram, it looks like the data is first read into the file, then gotten out and sent to the network. But that is not the only case. **A data flow diagram does NOT specify sequencing**.

For instance, it could be that once there is lots of data in the file, the 'typical' operation is to first get some data from the file and send it for storage somewhere else on the network (so it is not lost by overwriting).

Then, new data is read from the keyboard and overwrites the data just read from the disk file! This provides a stunning clear example of why you should NOT assume a DFD shows ordering.

Instead, you should think of it as a *plumbing diagram for data.* It illustrates:

- Where the data come in. Though the above diagram only shows one source, in a complicated system, the plumbing has several entry points.
- How the data is processed (split, sorted, modified, calculated).
- Where the data is stored for possible later use in constructing outputs.
- Finally, how the data is processed for output, and which of the many possibly outputs (screen, printer, network) that the data is ‘sinked’ to.

It does NOT illustrate ‘when’ (or ‘whether’ on last day of month) any particular event or process happens. It only shows ALL possible paths data might take say in an accounting system, on ANY day of the month.

4.10.1 Logical vs. Physical DFDs

Physical DFDs are used by systems analysts to document the flow of physical objects in an actual business. And example might be automobile parts flowing around a factory and being assembled. File icons are really warehouses or store rooms. Physical DFDs sometimes also show actual paperwork flowing from department to department.

If the paperwork flow is to be automated and implemented in a computer system, you then probably want to later draw a Logical DFD to illustrate how the data is flowing within the computer system, its files and its inputs and outputs.

Ask professor to draw an
example DFD

4.10.2 <u>DFD Processes</u>

Generally there are 4 kinds of DFD process bubbles:

- Change of nature - e.g. those that analyse an sales history and compute an annual compound growth rate.
- Change of composition - usually splits compound records so that some fields go out one flow and other fields leave via a third flow. Alternatively, two flows may join.
- Change of organization - typical examples are formatting a report, or sorting.

- Change of viewpoint - output is similar but is compared with others or corrected for errors.

4.10.3 DFD Advice

Students seem to have a hard time initially with DFDs. There are several reasons for this.

Firstly, students usually draw DFDs starting from the inputs and working toward the outputs. This quickly gets them bogged down in the user interface. But data you type into a user interface such as:

```
Main Menu:
1) Start Process
2) End Process
Enter your selection:
```

IS NOT DATA. The “1” or the “2” that the user will type will usually never be seen in the output (though they will be echoed by the keyboard process to the screen). They are used only to control the application; they are not data used in calculating the applications useful outputs! They therefore do NOT belong in a DATA flow diagram!

To get around this problem, I suggest you initially regard yourself as the consumer of the outputs. As a consumer you need to ‘suck’ data from the application. So, start drawing from the output end, thinking about how the data you suck out must have been constructed from the files and processes and inputs earlier in the DFD.

Secondly, students sometimes attach one and only one bubble to every file. This bubble controls reading and writing of the file. But reading and writing are two different functions/processes. I think the problem is students are thinking of the data in the file as an abstract data type (ADT) and that the bubble is the module that implements it. This is wrong. Generally, bubbles are not modules and do not represent the multiple functions that a module can usually contain.

Finally, when doing a Level 1 DFD, you might find it useful to add one file for each object entity in your data model. If you add a bubble called ‘user interface’ it is not to show control flow or processing, but just the splitting of the input data to route it to the appropriate places in your data plumbing diagram. Occasionally, you may find it useful to put a bubble in for each top level menu item. This is called operation set partitioning. When decomposing each of those high level menu bubbles to Level 2 detail, draw a sub-bubble for each individual menu leaf item. Note that you don’t have to do this, but it is one possible way to use DFDs which were conceived before menus (!), for analyzing a menu oriented application.

4.10.4 Decomposing DFDs

When you first start analysis, one of the first things you do is ask about what are the inputs and outputs. This should be immediately documented in a top level data flow diagram where there is only one process bubble: the system as a whole. This top level DFD is sometimes also called a **Context Diagram or Level 0 DFD**. See [Pressman97] page 324 for an example. A Context Diagram basically illustrates all the inputs and outputs of the system.

During analysis you hierarchically decompose the context diagram to show what goes on inside the bubble in the Context Diagram. Such a Level 1

DFD is shown on page 326 of [Pressman97]. Note that there are just as many arrows entering and exiting the Level 1 DFD as there were entering and exiting the bubble in the Context diagram. In essence, you could draw the context diagram bubble as a very big circle on the Level 1 DFD just inside the squares. I personally don't think Pressman's Level 1 DFD should even have the squares, just labelled incoming flow arrows.

Page 311 of [Pressman97] illustrates that this decomposition can continue indefinitely, until you feel you have an adequate understanding of the required system and can illustrate this understanding to those who need to know. For instance, a further decomposition of the ‘monitor sensors’ bubble in the Level 1 DFD is given by Pressman on Page 327. Note that on Page 311, a hierarchical numbering scheme is give to each process, and included in the data dictionary so you can easily find a process in a myriad of diagrams that might be on file for a system design.

The assigned readings in [Pressman97] also cover starting on Page 315 two different extensions to the traditional DFD Structured Design process. These are called ‘real-time’ extensions but THEY HAVE NOTHING TO DO WITH REAL-TIME SYSTEM OR REAL TIME PROCESSING!   It is just that real-time control systems are often carefully designed, and the designers are often interested in specifying more than just hierarchical plumbing. Instead they want to specify sequential ordering of which process/function is activated in what order.

You do not have to memorize the real time extensions, but you may be asked to write a paragraph on why extensions are needed and that they are often implemented using finite state machines to trigger the processes in the correct order (or if the order is independent of any kind of state or mode, then a process activation table can be used). So please at least read about the extensions. Can you find anything in the readings that talks about real time criticality deadlines?